Master 2 LITL
Programmation pour le TAL (SLT0905V)
Les interfaces et les *handlers*

Franck Sajous/CLLE-ERSS

http://fsajous.free.fr/

Les interfaces

C'est quoi ?

- Comme une classe abstraite, sans attributs
- Uniquement des signatures de méthodes
- Modélise un « comportement » commun (de classes de natures potentiellement différentes)

Exemples (réels)

- *Comparable*: `int compareTo (Object o)`
- *Cloneable*: `Object clone ()`
- *Serializable*:
  - `void writeObject (ObjectOutputStream out)`
  - `void readObject (ObjectInputStream in)`

Les interfaces : autres exemples

des trucs...

- dont on peut donner une représentation textuelle, HTML, CSV
- qui ont une couleur et sur lesquels on peut appliquer des effets

```
public interface TXTprintable
{
    public String toTXT ();
}

public interface CSVprintable
{
    public String toCSV ();
}

public interface HTMLprintable
{
    public String toHTML();
}

public interface Colored
{
    public Color getColor ();
    public void  setColor (Color c);
    public void  highlight ();
    public void  solarize ();
    public void  toGreyScale ();
}
```

Les interfaces : autres exemples

des trucs...

- dont on peut donner une représentation textuelle, HTML, CSV
- qui ont une couleur et sur lesquels on peut appliquer des effets

```
public interface TXTprintable
{
    public String toTXT ();
}

public interface CSVprintable
{
    public String toCSV ();
}

public interface HTMLprintable
{
    public String toHTML();
}

public interface Colored
{
    public Color getColor ();
    public void  setColor (Color c);
    public void  highlight ();
    public void  solarize ();
    public void  toGreyScale ();
}
```

implements

Une classe peut *implémenter* une interface (ou plusieurs)

```
public class C
implements TXTprintable, HTMLprintable
{
     public String toString () {
       return "...";
     }
     public String toHTML () {
       return "...";
     }
}
```

Les interfaces : autres exemples

```
public class ManualAnnotation
          implements HTMLprintable, TXTprintable, Colored
{
    private Token  token;
    private String annotationText;
    private Date   annotationDate;
    private Color  annotationColor;

    public String toTXT() {
        return "...";
    }
    public String toHTML() {
        return "...";
    }
    public Color getColor() {
        return annotationColor;
    }
    public void setColor(Color c) {
        annotationColor = c;
    }
    public void highlight() {
        // color.highlight();
    }
    public void solarize() {
        // TODO implement method solarize
    }
    public void toGreyScale() {
        // ...
    }
}
```

```
public class IndirectDependency
                extends SyntacticDepency
                implements TXTprintable, HTMLprintable
{
    private Token pivot;

    public Token getPivot()
    {
        return pivot;
    } // getPivot()
    public void setPivot(Token p)
    {
        pivot = p;
    } // setPivot(Token p)
    public String toTXT ()
    {
        return "<" + getGovernor().toTXT()
                + ";" + getDepLabel() + "/"
                + getPivot().getLemma()
                + ";" + getDependent().toTXT();
    } // String toTXT ()
    public String toHTML ()
    {
        return "<" + getGovernor().toHTML()
                + ";" + getDepLabel() + "/"
                + getPivot().getLemma()
                + ";" + getDependent().toHTML();
    } // String getTextRepresentation ()
} // class IndirectDependency
```

Les interfaces : aussi un type

```
List<Colored> thingsHavingAcolor = new ArrayList<Colored> ();
// ... creating Colored instances ...
// ... and adding them to the list

Iterator<Colored> itCol = thingsHavingAcolor.iterator();

while (itCol.hasNext())
{
    Colored coloredStuff = itCol.next();
    coloredStuff.solarize();
}
```

Les interfaces : aussi un type

```
List<Colored> thingsHavingAcolor = new ArrayList<Colored> ();
// ... creating Colored instances ...
// ... and adding them to the list

Iterator<Colored> itCol = thingsHavingAcolor.iterator();

while (itCol.hasNext())
{
    Colored coloredStuff = itCol.next();
    coloredStuff.solarize();
}
```

On ne connait pas la classe X dont `coloredStuff` est l'instance.

On sait juste que cette classe implémente l'interface `Colored`.

Les interfaces : aussi un type

```
List<Colored> thingsHavingAcolor = new ArrayList<Colored> ();
// ... creating Colored instances ...
// ... and adding them to the list

Iterator<Colored> itCol = thingsHavingAcolor.iterator();

while (itCol.hasNext())
{
    Colored coloredStuff = itCol.next();
    coloredStuff.solarize();
}
```

On ne connait pas la classe X dont `coloredStuff` est l'instance.
On sait juste que cette classe implémente l'interface `Colored`.

À l'itération suivante, `coloredStuff` sera une instance de la classe Y (égale à, ou différente de X), qui implémente aussi `Colored`.

Éviter l'héritage multiple

`implements`

- Une classe peut *implémenter* une interface (ou plusieurs)

Éviter l'héritage multiple

implements

- Une classe peut *implémenter* une interface (ou plusieurs)
- S'il s'agit uniquement de signatures de méthodes, pourquoi ne pas utiliser des classes abstraites sans membres ?

```
      abstract class
public ~~interface~~ Colored
{
    public Color getColor ();
    public void  setColor (Color c);
    public void  highlight ();
    public void  solarize ();
    public void  toGreyScale ();
}
```

Éviter l'héritage multiple

implements

- Une classe peut *implémenter* une interface (ou plusieurs)
- S'il s'agit uniquement de signatures de méthodes, pourquoi ne pas utiliser des classes abstraites sans membres ?

```
      abstract class
public ~~interface~~ Colored
{
    public Color getColor ();
    public void  setColor (Color c);
    public void  highlight ();
    public void  solarize ();
    public void  toGreyScale ();
}
```

TokenHTML

Une classe qui *est un* Token et a des trucs en plus (une police, une position, un style, etc.).

→ `public class TokenHTML extends Token`

Éviter l'héritage multiple

implements

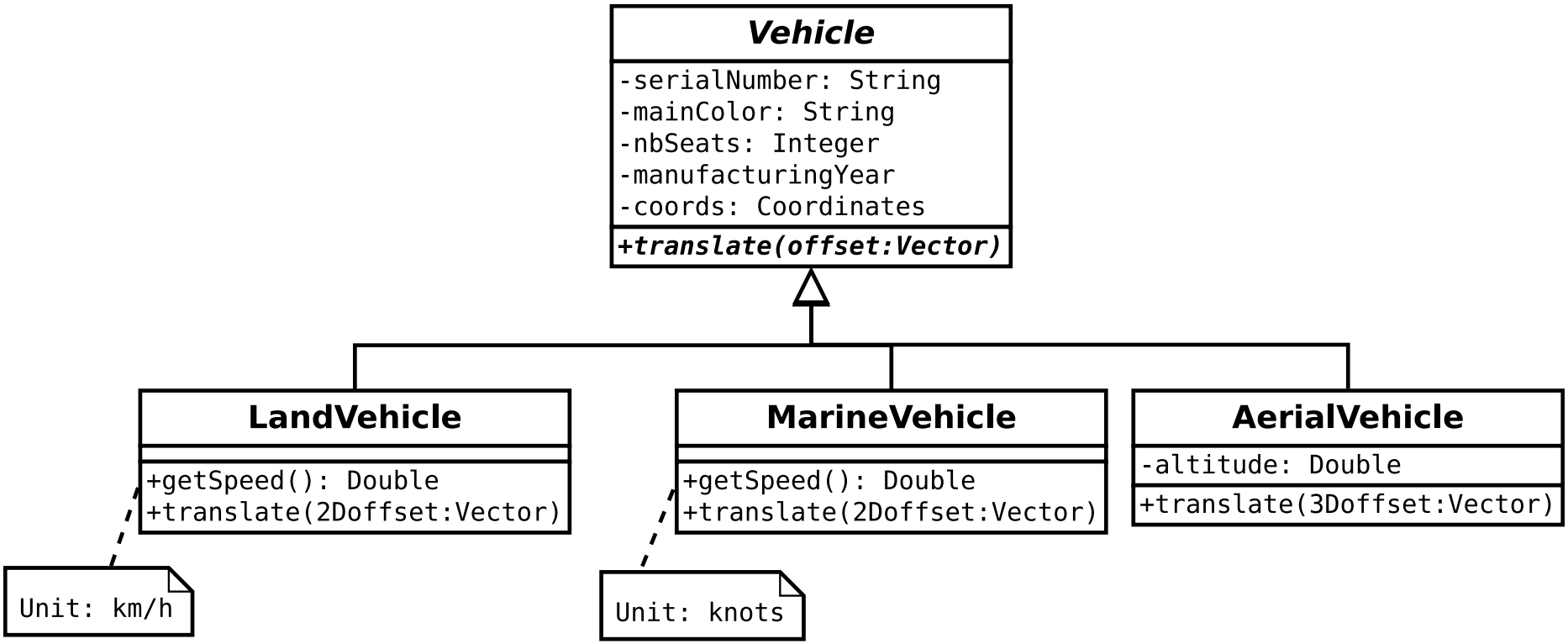
- Une classe peut *implémenter* une interface (ou plusieurs)
- S'il s'agit uniquement de signatures de méthodes, pourquoi ne pas utiliser des classes abstraites sans membres ?

```
      abstract class
public ~~interface~~ Colored
{
    public Color getColor ();
    public void  setColor (Color c);
    public void  highlight ();
    public void  solarize ();
    public void  toGreyScale ();
}
```

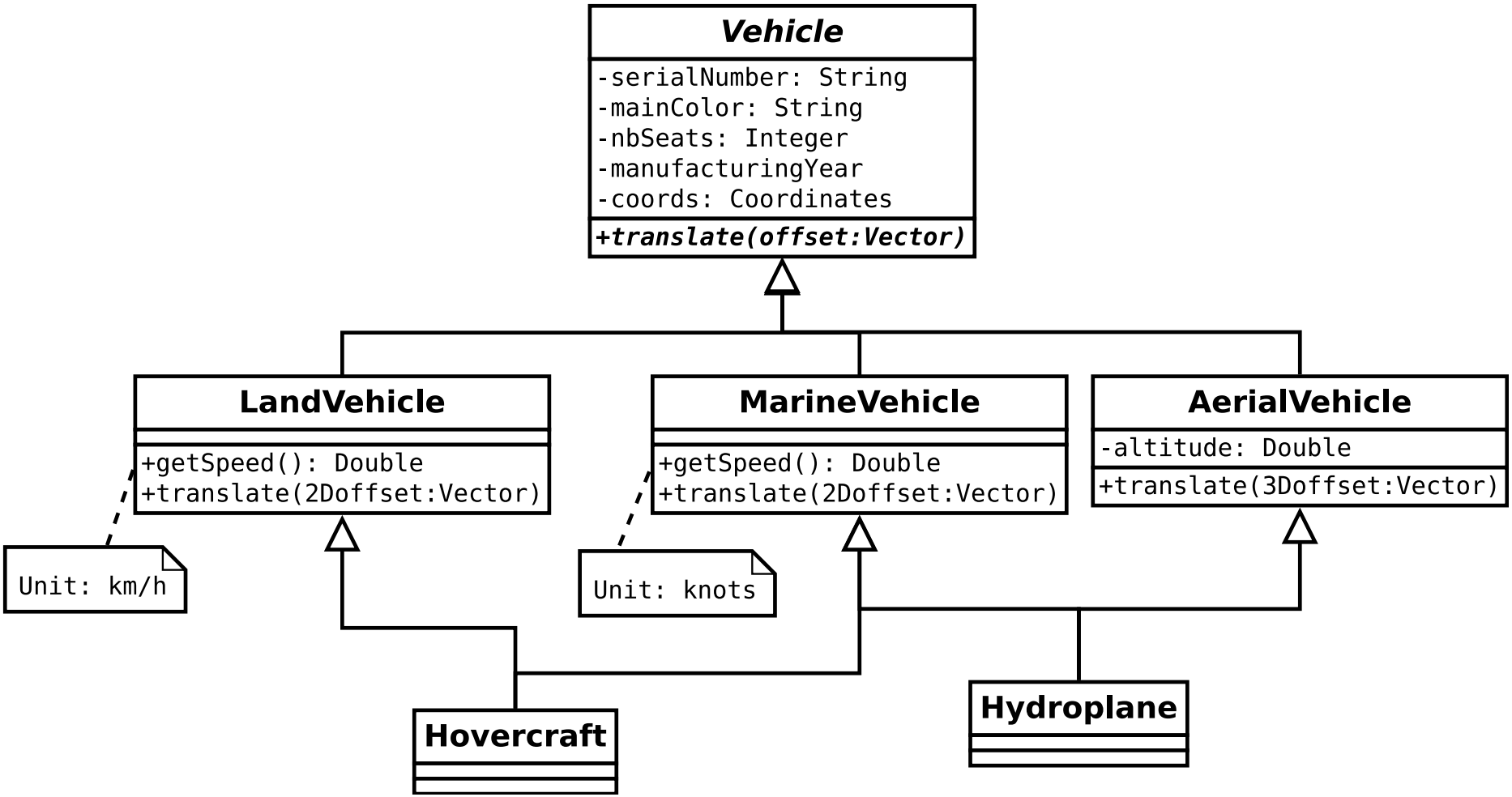
TokenHTML

Une classe qui *est un* Token et a des trucs en plus (une police, une position, un style, etc.).
→ `public class TokenHTML extends Token`
Un TokenHTML « a » une couleur. Comment modéliser le fait qu'un TokenHTML « *a le comportement* » (« *fournit les services* ») défini(s) dans Colored ?

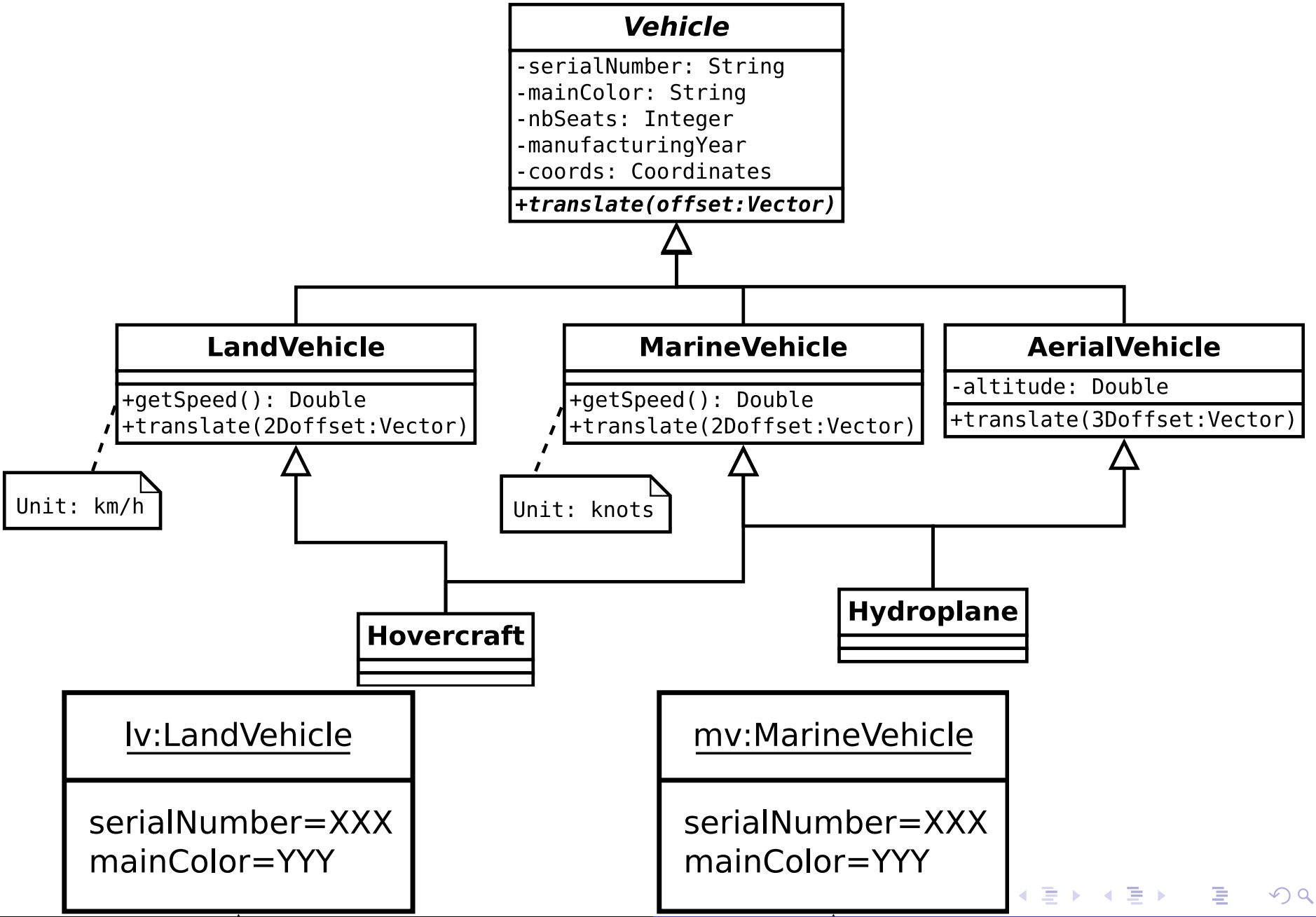
Éviter l'héritage multiple et le problème du « diamant »



Éviter l'héritage multiple et le problème du « diamant »



Éviter l'héritage multiple et le problème du « diamant »



Éviter l'héritage multiple et le problème du « diamant »