ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 31 janvier 2011

Résumé

– Interface
– Type abstrait de données

*. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Interface 1

En programmation orientée objet, l'interface d'une classe désigne l'ensemble des méthodes et variables publiques de la classe.

Par exemple, si la classe **Time** possède trois méthodes publiques, **getHours()**, **getMinutes()** et getSeconds(), et aucune autre méthode ou variable publique, alors l'interface de la classe (au sens large) est : **getHours()**, **getMinutes()** et **getSeconds()**.

Problème 1

Parfois, l'utilisation de l'héritage simple n'est pas justifiée ou n'est pas suffisante. Les trois exemples qui suivent visent à démontrer ceci.

Problème 1 : écrivez une méthode (polymorphique) afin de trier les valeurs d'un tableau.

Il existe une variété d'algorithmes de tri, tels que ≪bubble sort≫, ≪selection sort≫, ≪quick sort≫, etc.

Qu'ont en commun ces algorithmes ? (attendez avant de répondre)

Afin de rendre l'exemple plus concis, nous allons nous restreindre à traiter des tableaux de taille 2.

Ainsi, l'algorithme de tri sera simple : si la valeur à la position 0 est plus grande que celle à la position 1 alors il faut les échanger, sinon il n'y a rien à faire, le tableau est déjà ordonné.

Un algorithme plus général sera présenté par la suite.

Voici une méthode pour trier un tableau de deux entiers.

```
public static void sort2( int[] a ) {
    if ( a[ 0 ] > a[ 1 ] ) {
        int tmp = a[ 0 ];
        a[ 0 ] = a[ 1 ];
        a[ 1 ] = tmp;
    }
}
```

$\Rightarrow$ Quels sont les changements nécessaires afin que cette méthode puisse traiter un tableau de références vers des objets de la classe **Time** ?

1) Il faut changer le type du paramètre et 2) remplacer l'opérateur de comparaison par un appel à la méthode **after(Time other)**.

```
public static void sort2( Time[] a ) {
    if ( a[ 0 ].after( a[ 1 ] ) ) {
        Time tmp = a[ 0 ];
        a[ 0 ] = a[ 1 ];
        a[ 1 ] = tmp;
    }
}
```

Quels sont les changements nécessaires afin que cette méthode puisse traiter un tableau d'objets de type **Shape** ?

1) Il faut changer le type du paramètre et 2) remplacer l'appel à methode **after(Time other)**, par un appel à une méthode permettant de comparer deux formes géométriques entre elles.

```
public static void sort2( Shape[] a ) {
    if ( a[ 0 ].compareTo( a[ 1 ] ) > 0) {
        Shape tmp = a[ 0 ];
        a[ 0 ] = a[ 1 ];
        a[ 1 ] = tmp;
    }
}
```

Qu'ont en commun ces algorithmes ?

Afin de faire le tri des valeurs d'un tableau, il suffit de savoir comparer deux objets de même type entre eux.

Le tri d'objets est une opération fréquente, il serait donc avantageux d'avoir une méthode fonctionnant pour tout type d'objets ayant une méthode afin de comparer deux objets.

Quels sont les exigences/besoins/spécifications/types/opérations ?

```
static void sort2( __________[] as ) {

   if ( as[ 0 ].__________( as[ 1 ] ) > 0) {

       __________ tmp;
       tmp = as[ 0 ];
       as[ 0 ] = as[ 1 ];
       as[ 1 ] = tmp;
   }
}
```

Tous ont en commun le fait qu'il faille comparer des paires de valeurs afin de déterminer si les valeurs sont en ordre ou pas.

1. Il faut une méthode afin de comparer deux objets ;
2. Le nom de la méthode devrait être le même pour toutes les classes qui souhaitent utiliser cette méthode de tri ;
3. L'implémentation de cette méthode sera spécifique à chaque classe.

Qui suis-je ?

Solution 1 : la superclasse Comparable

Voici une solution **insatisfaisante** que nous revisiterons plus tard.

Nous tenterons de résoudre ce problème à l'aide des outils que nous avons à notre disposition, programmation orientée objet et héritage, pour nous apercevoir qu'il nous faut un nouveau concept.

```
public ____1___ class Comparable {
   public ____1___ int compareTo( _____2____ obj );
}
```

Remplacez les éléments manquants.

L'élément 1 est le mot clé ≪ abstract ≫. En effet, l'implémentation de la méthode **compareTo** dépend du type des objets.

```
public abstract class Comparable {
   public abstract int compareTo( _____2____ other );
}
```

L'élément 2 est plus difficile à trouver.

```
public class Array {

  public static void sort2( Comparable[] as ) {
                           ----------
     if ( as[ 0 ].compareTo( as[ 1 ] ) > 0 ) {
                  ---------
        Comparable tmp;
        ----------
        tmp = as[ 0 ];
        as[ 0 ] = as[ 1 ];
        as[ 1 ] = tmp;
     }
  }
}
```

Si vous aviez répondu **Comparable**, cette réponse est tout à fait acceptable.

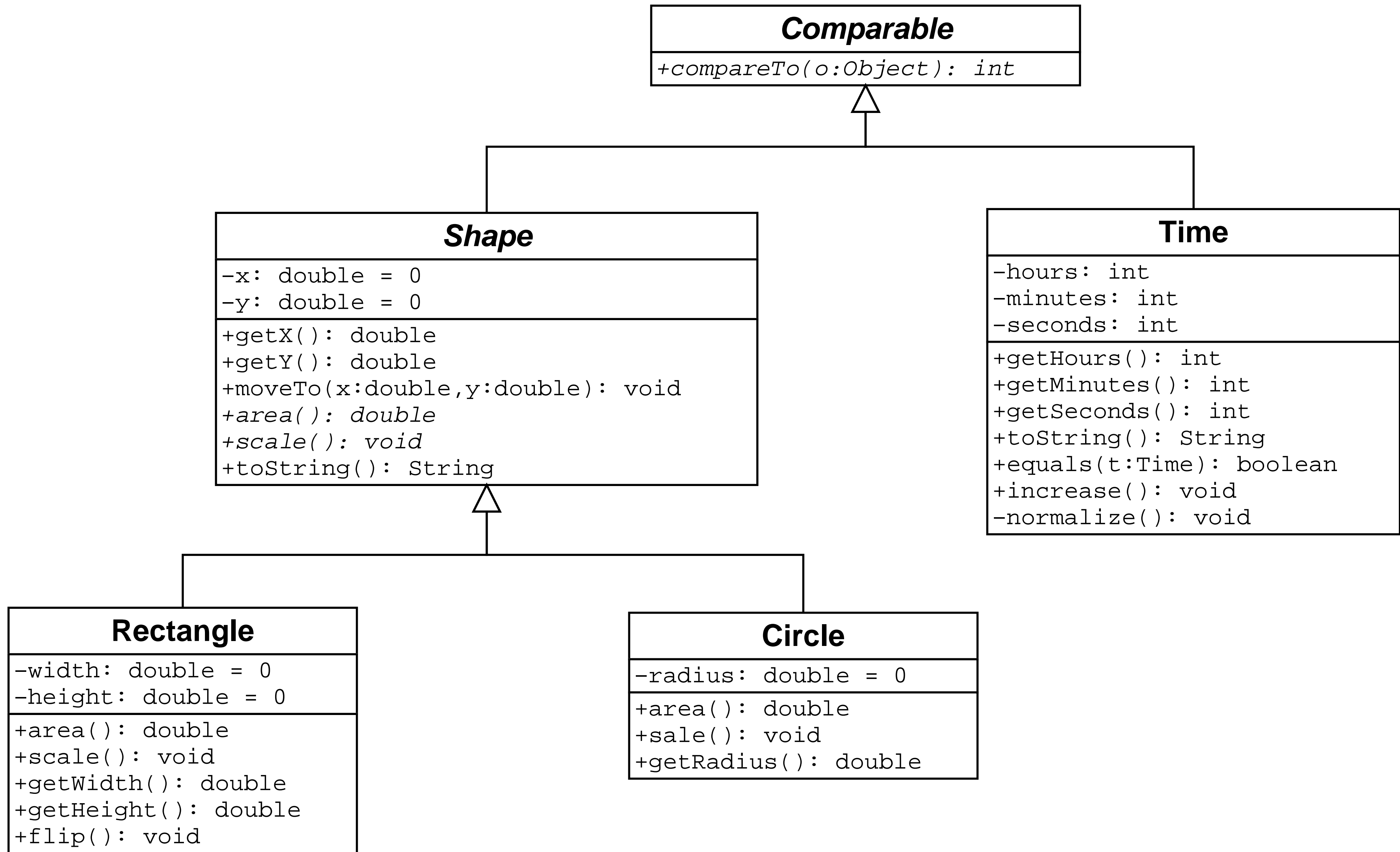
Cependant, l'utilisation du type **Object** rend la méthode plus générale.

```
abstract class Comparable {
   public abstract int compareTo( Object obj );
}
```

Je propose donc la création d'une classe abstraite, appelée **Comparable**, n'ayant qu'une seule méthode, nommée **compareTo(Object other)**.

Cette méthode doit bien évidemment être abstraite, puisque la comparaison de deux objets dépend de leur implémentation.

Toute classe désirant bénéficier de la méthode polymorphique **sort2** devra être une sous-classe de la classe **Comparable**.


Comparable
+compareTo(o:Object): int
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): double
+moveTo(x:double,y:double): void
+area(): double
+scale(): void
+toString(): String
Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
Rectangle
-width: double = 0
-height: double = 0
+area(): double
+scale(): void
+getWidth(): double
+getHeight(): double
+flip(): void
Circle
-radius: double = 0
+area(): double
+sale(): void
+getRadius(): double

Pour la classe **Shape**, on écrira,

```
public abstract class Shape extends Comparable {

    // ....

    public int compareTo( Shape other ) {
        int result;
        if ( area() < other.area() )
            result = -1;
        else if ( area() == other.area() )
            result = 0;
        else
            result = 1;
        return result;
    }
}
```

Oups, le compilateur n'aimera pas ça ! Pourquoi ?

Le problème, c'est la signature ≪**public int compareTo(Shape obj)**≫.

Quel est le problème ?

Quel serait le message d'erreur ?

≪ `Shape is not abstract and does not override abstract method compareTo(java.lang.Object) in Comparable` ≫

```
public abstract class Shape extends Comparable {

    // ....

    public int compareTo( Object other ) {
        int result;
        if ( area() < other.area() )
            result = -1;
        else if ( area() == other.area() )
            result = 0;
        else
            result = 1;
        return result;
    }
}
```

Attention ! La tentation d'écrire ≪public int compareTo(Shape obj)≫ sera très grande, il vous faudra résister !

Ce n'est pas terminé ! La compilation de cette classe produira une erreur, quelle est-elle ?

```
public class Shape extends Comparable {
    ....
    public int compareTo( Object other ) {

        int result;

        if ( area() < other.area() )
            result = -1;
        else if ( area() == other.area() )
            result = 0;
        else
            result = 1;

        return result;
    }
}
```

Le paramètre est de type **Object**, la méthode **area()** n'est pas définie dans la classe **Object**. Solution ?

```
public class Shape extends Comparable {
    ....
    public int compareTo( Object other ) {

        int result;

        if ( area() < other.area() )
            result = -1;
        else if ( area() == other.area() )
            result = 0;
        else
            result = 1;

        return result;
    }
}
```

Le paramètre est de type **Object**, la méthode **area()** n'est pas définie dans la classe **Object**. Solution :

```
public class Shape extends Comparable {
    ....
    public int compareTo( Object o ) {
        Shape other = (Shape) o;
        int result;

        if ( area() < other.area() )
            result = -1;
        else if ( area() == other.area() )
            result = 0;
        else
            result = 1;

        return result;
    }
}
```

Ensuite, la classe **Time** devra fournir une implémentation pour la méthode **compareTo** si on souhaite utiliser la méthode **Array.sort2** afin de trier un tableau d'objets **Time**.

```
public class Time extends Comparable {
  ...
  public int compareTo(Object obj) {
      Time other = (Time) obj;
      int result;
      if ( timeInseconds < other.timeInSeconds ) {
         result = -1;
      } else if ( timeInSeconds == other.timeInSeconds ) {
         result = 0;
      } else {
         result = 1;
      }
      return result;
  }
}
```

Voici l'implémentation d'une méthode polymorphique pour le tri d'objets ≪comparables≫.

```
public class Array {

    public static void sort2( Comparable[] a ) {
        if ( a[ 0 ].compareTo( a[ 1 ] ) > 0 ) {
            Comparable tmp = a[ 0 ];
            a[ 0 ] = a[ 1 ];
            a[ 1 ] = tmp;
        }
    }

}
```

Test :

```
Time[] times = new Time[ 2 ];

Times[ 0 ] = new Time( ... );
Times[ 1 ] = new Time( ... );

Array.sort2( times );

Shape[] shapes = new Shape[ 2 ];

shapes[ 0 ] = new Circle( ... );
shapes[ 1 ] = new Rectangle( ... );

Array.sort2( shapes );
```

Problème résolu ! Pour l'instant . . .

Problème 2

Problème 2 : concevoir une méthode polymorphique permettant l'affichage de tous les éléments d'un tableau (des formes, des boutons, des ballons, . . .).

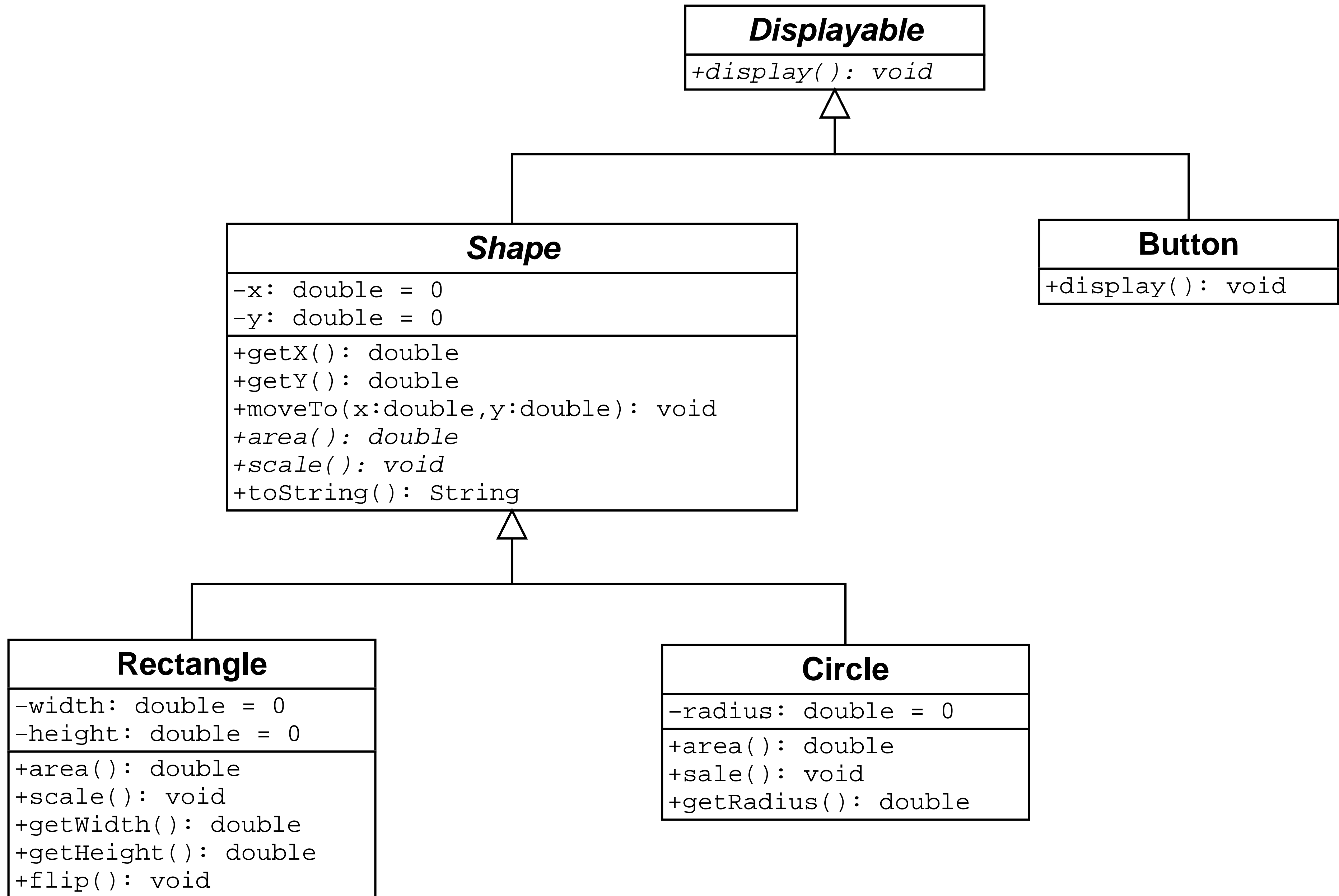
```
public class Graphics {

     public static void displayAll( ________[] as ) {
          for ( int i=0; i<as.length; i++ ) {
               as[ i ].display();
          }
     }
}
```

Quel sera le type des éléments du tableau ?

```
public ________ class Displayable {
    public ________ void display();
}
```

Trouvez l'élément manquant.

```
public abstract class Displayable {
    public abstract void display();
}
```


Displayable
+display(): void
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): double
+moveTo(x:double,y:double): void
+area(): double
+scale(): void
+toString(): String
Button
+display(): void
Rectangle
-width: double = 0
-height: double = 0
+area(): double
+scale(): void
+getWidth(): double
+getHeight(): double
+flip(): void
Circle
-radius: double = 0
+area(): double
+sale(): void
+getRadius(): double

Utilisation : toute classe désirant utiliser **Graphics.displayAll(Displayable[] as)** devra :

1. Être une sous-classe de la classe **Displayable** ;
2. Implémenter la méthode **display()**.

Problème résolu ! Pour l'instant . . .

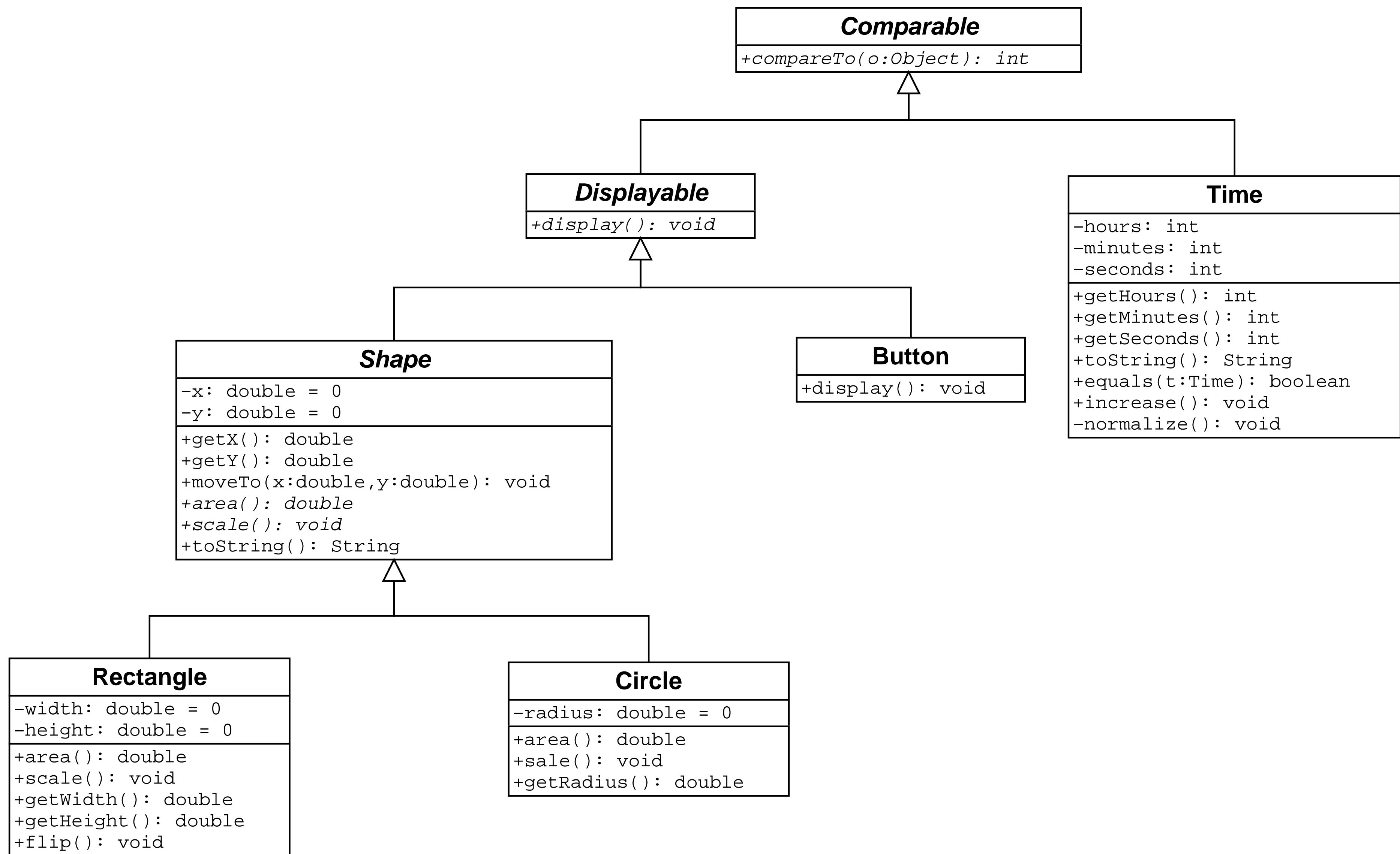
Problème 3

Problème 3 : nous souhaitons maintenant que la classe **Shape** soit à la fois **Comparable et Displayable** !

Quelles sont les solutions possibles ?

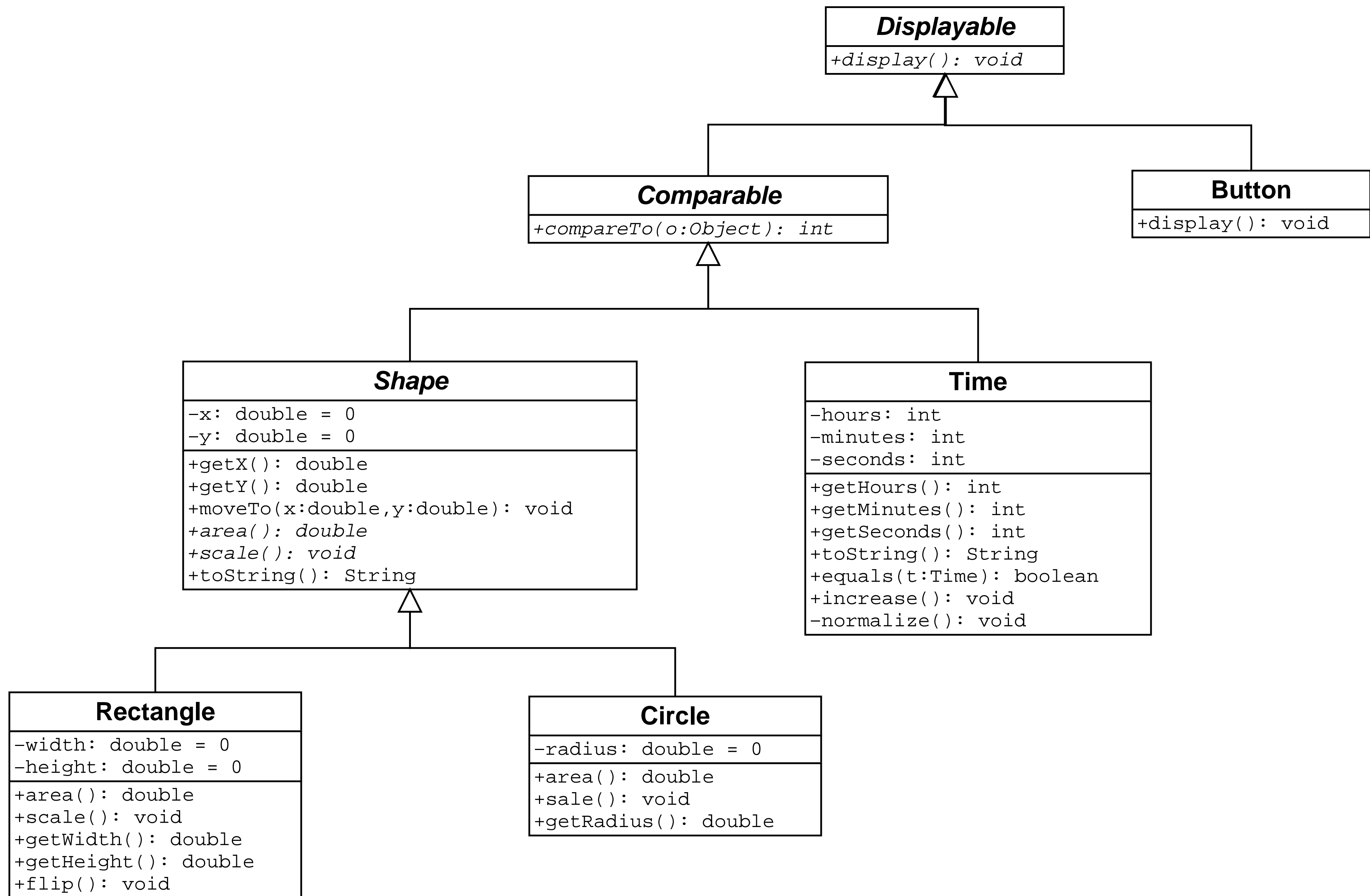
Une solution possible serait que la classe **Displayable** soit une sous-classe de la classe **Comparable**, forçant ainsi toute sous-classe de la classe **Displayable** à implémenter la méthode **display** et la méthode **compareTo**.

Qu'en pensez-vous ?


Comparable
+compareTo(o:Object): int
Displayable
+display(): void
Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): double
+moveTo(x:double,y:double): void
+area(): double
+scale(): void
+toString(): String
Button
+display(): void
Rectangle
-width: double = 0
-height: double = 0
+area(): double
+scale(): void
+getWidth(): double
+getHeight(): double
+flip(): void
Circle
-radius: double = 0
+area(): double
+sale(): void
+getRadius(): double

Le problème avec cette organisation des classes est qu'elle force toute sous-classe de **Displayable** à implémenter aussi la méthode **compareTo**.

Il existe peut-être des situations pour lesquelles ce n'est pas souhaitable : la classe button par exemple.

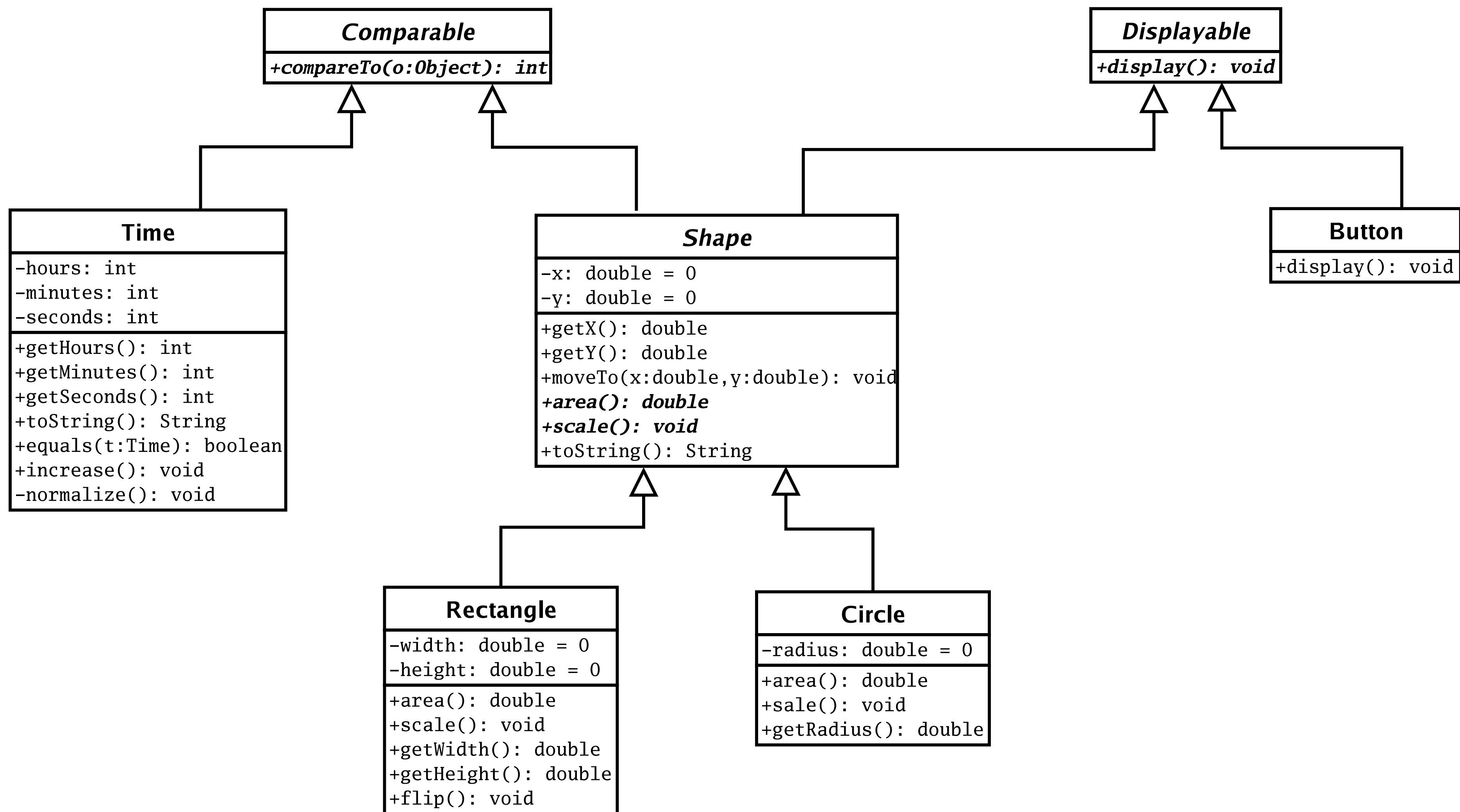

Displayable
+display(): void
Comparable
+compareTo(o:Object): int
Button
+display(): void
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): double
+moveTo(x:double,y:double): void
+area(): double
+scale(): void
+toString(): String
Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
Rectangle
-width: double = 0
-height: double = 0
+area(): double
+scale(): void
+getWidth(): double
+getHeight(): double
+flip(): void
Circle
-radius: double = 0
+area(): double
+sale(): void
+getRadius(): double

On pourrait ré-organiser la hiérarchie de sorte que la classe **Displayable** soit au sommet de l'arbre.

Ça ne ferait que déplacer le problème.

L'arbre d'héritage simple semble la cause du problème.

Solution ?

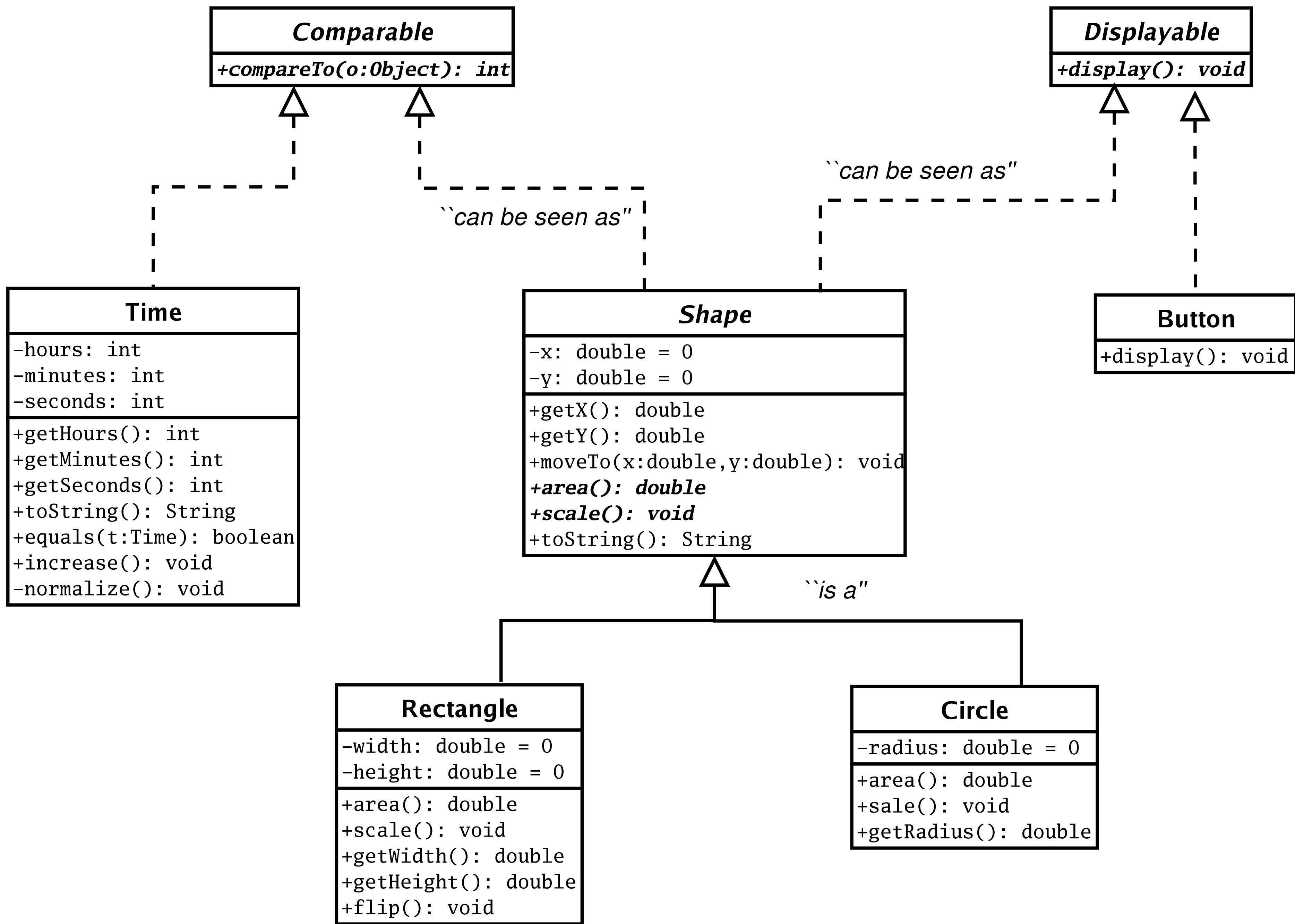

Comparable
+compareTo(o:Object): int
Displayable
+display(): void
Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): double
+moveTo(x:double,y:double): void
+area(): double
+scale(): void
+toString(): String
Button
+display(): void
Rectangle
-width: double = 0
-height: double = 0
+area(): double
+scale(): void
+getWidth(): double
+getHeight(): double
+flip(): void
Circle
-radius: double = 0
+area(): double
+sale(): void
+getRadius(): double

Java ne permet pas l'héritage multiple.

```
class Shape extends Comparable, Displayable {
   ...
}
```

C'est un cul-de-sac !

Java propose une approche différente, l'interface (et ce concept permet d'implémenter des relations de type ≪can be seen as≫ plutôt qu'≪is a≫ dans le cas de l'héritage).


Comparable
+compareTo(o:Object): int
Displayable
+display(): void
``can be seen as"
``can be seen as"
Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): double
+moveTo(x:double,y:double): void
+area(): double
+scale(): void
+toString(): String
Button
+display(): void
``is a"
Rectangle
-width: double = 0
-height: double = 0
+area(): double
+scale(): void
+getWidth(): double
+getHeight(): double
+flip(): void
Circle
-radius: double = 0
+area(): double
+sale(): void
+getRadius(): double

Interface

La déclaration d'une interface ressemble à celle d'une classe.

Elle débute par le mot clé **interface** (et non **class**), suivi d'un identificateur (le nom de l'interface), suivi par le corps de l'interface.

```
public interface Comparable {
    public abstract int compareTo( Object o );
}
```

La définition est mise dans un fichier du même nom que l'interface, puis elle est compilée afin de produire un fichier **.class**.

Interface

L'interface contient :
- Constantes ;
- Méthodes abstraites.

Tout comme une classe abstraite, on ne peut créer une instance d'une interface.

À la différence d'une classe abstraite, on ne peut y mettre de méthodes concrètes ou des variables.

Comparable

L'interface **Comparable** fait partie de la librairie standard de Java.

```
public interface Comparable {

    // Compare cet objet et celui passé en paramètre
    // afin d'en déterminer l'ordre.  Retourne une
    // valeur négative, zéro ou positive selon que
    // cet object est plus petit, égal ou plus grand
    // que l'objet specifié.

    public abstract int compareTo( Object o );
}
```

Cette interface définit l'ordre naturel des éléments d'une classe.

Nous avons définit un type abstrait, il lui faut maintenant une implémentation.

```
public class Shape implements Comparable {


}
```

Nouveau mot-clé, **implements**. Nous dirons que la classe **Shape réalise** l'interface **Comparable**.

Mais encore ! La classe **Shape** doit fournir une implémentation pour la méthode **int compareTo(Object o)**.

De façon générale, lors qu'une classe réalise une interface est doit implémenter toutes méthodes qui s'y trouvent.

Qu'avons-nous gagné ? Une classe peut réaliser plus d'une interface.

```
public class Shape implements Comparable, Displayable {

}
```

Ça veut tout simplement dire que la classe **Shape** doit fournir une implémentation pour la méthode **compareTo** ainsi qu'une implémentation pour la méthode **display**.

Mais encore ! À quoi ça sert ?

```
Comparable a;  // valide?
Comparable b;  // valide?

a = ...;
b = ...;

if ( a.compareTo( b ) > 0 ) {
   ...
}
```

Le nom d'une interface peut-être utilisé pour la déclaration d'une variable référence.

```
Comparable o;
```

Que peut-on y mettre ?

Est-ce un énoncé valide ?

```
Comparable o = new Comparable();
```

Cet énoncé n'est pas valide ?

```
Comparable o = new Comparable();
```

Ceux-ci le sont.

```
Comparable o;
Circle c = new Circle( 0, 0, 1 );
o = c;
o = new Circle( 10, 5, 123 );
```

On peut y mettre la référence d'un objet dont la classe implémente l'interface.

```
o.getX(); // n'est pas valide
c.getX(); // est valide
```

Question piège, est-ce que ces énoncés sont valides ?

```
Comparable o;
o = new Circle( 0, 0, 1 );
o.toString();
```

Concrètement, une référence est toujours une référence vers un objet et tout objet en Java possède les caractéristiques de la classe **Object**.

Exemple d'utilisation

La classe **String** réalise l'interface **Comparable**.

On peut donc utiliser la méthode de tri de la classe **Arrays**.

```
import java.util.Arrays;

public class Test {
    public static void main( String[] args ) {

        Arrays.sort( args );

        for ( int i=0; i<args.length; i++ )
            System.out.println( args[i] );
    }
}
```

Exemple d'utilisation

```
> java Test using interfaces in Java
Java
in
interfaces
using
```

De même, la définition de toute classe peut être modifiée afin de réaliser l'interface **Comparable** et ainsi bénéficier des méthodes prédéfinies qui nécessitent l'utilisation du comparateur **compareTo**.

```
public class Time implements Comparable { ... }
```

Il faut alors implémenter la méthode **compareTo**.

Il sera alors possible d'utiliser la méthode **sort()**.

```
Time[] ts = new Time[ 100 ];
...
Arrays.sort( ts );
```

```
public class SortAlgorithms {
    public static void selectionSort( Comparable a[] ) {
        for ( int i = 0; i < a.length; i++ ) {
            int min = i;
            // trouvez l'element le plus petit dans la
            // portion non triee du tableau
            for ( int j = i+1; j < a.length; j++ )
                if ( a[j].compareTo( a[ min ] ) < 0 )
                    min = j;
            // echanger cet element et celui en position i
            Comparable tmp = a[ min ];
            a[ min ] = a[ i ];
            a[ i ] = tmp;
        }
    }
}
```

⇒ Voir www.cs.ubc.ca/spider/harrison/Java

Implémenter de multiples interfaces

```
class A implements B, C, D {
   ...
}
interface B {
  public abstract void b1();
  public abstract void b2();
}
interface C {
  public abstrat void c1();
}
interface D {
  public abstract void d1();
  public abstract void d2();
  public abstract void d3();
}
```

$\Rightarrow$ Il n'y a pas d'héritage multiple en Java, mais il est possible et courant d'implémenter plusieurs interfaces.

L'interface ressemble à une classe abstraite.

Une interface ne peut contenir que des méthodes abstraites.

Une classe abstraite peut contenir des méthodes concrètes.

Interfaces et compilation

Considérez le scénario suivant : l'interface **Comparable**, la classe **SortAlgorithms** ayant un algorithme de tri dont le paramètre est de type **Comparable**, et la classe **Shape** (réalisant l'interface **Comparable**).

L'interface définit un contrat. (**Comparable**)

Une classe qui déclare une (des) variable(s) de type interface doit utiliser ces variables de façon consistante par rapport au type de l'interface, c.-à-d. ne peut qu'utiliser des méthodes de l'interface. (**SortAlgorithms**)

Une classe réalisant une interface doit fournir une implémentation pour chacune des méthodes déclarées par l'interface. (**Shape**)

La classe **SortAlgorithms** peut être compilée en l'absence d'une implémentation.

Interfaces vs classes abstraites

Ces deux concepts se ressemblent beaucoup, tous deux permettent la déclaration de méthodes abstraites et ne peuvent servir à la création d'instances.

Comment se distinguent les interfaces et les classes abstraites ?

- L'interface ne contient que des méthodes abstraites (c.-à-d. pas d'implémentation), si une classe abstraite ne contient aucun détail d'implémentation, il serait peut-être mieux d'en faire une interface ;
- Si le problème à modéliser requiert l'héritage multiple, il faut alors utiliser des interfaces ;
- Si le problème nécessite l'utilisation de méthodes abstraites et concrètes, il faut utiliser une classe abstraite ;
- L'interface définit la relation ≪can be seen as≫ ;
- L'héritage définit la relation ≪is a≫.

Interfaces

Le concept d'interface est utile pour les situations où il y a plusieurs implémentations possibles.

Considérez les deux implémentations de la classe **Time**.

Ces deux implémentations n'ont rien en commun, si ce n'est qu'au niveau de leur ≪interface≫.

```
public interface Time {
    public int getHours();
    public int getMinutes();
    public int getSeconds();
}

Time t1 = new Time1( 12, 10, 0 );
Time t2 = new Time2( 14, 15, 0 );
```

Types génériques et types paramétrés

Définir un type générique.

Un **type générique** est un **type référence** ayant un ou plusieurs paramètres de type.

Un **type générique**, c'est une **interface** ou une **classe** ayant un ou plusieurs paramètres de type.

Définir un type générique

```
public interface Comparable<T> {
    public int compareTo( T other );
}
```

Créer un type paramétré

```
public class Employee implements Comparable<Employee> {

    private int uid;

    public Employee( int uid ) {
        this.uid = uid;
    }

    public int getUid() { return uid; }

    public void setUid( int value ) { uid = value; }

    public int compareTo( Employee other ) {
        return uid - other.uid;
    }

}
```

Méthodes génériques

En plus des types génériques (interfaces et classes ayant des paramètres formels de type), les méthodes peuvent avoir un paramètre de type.

Méthode générique

```
public class Utils {
    public static < T extends Comparable<T> > T max( T a, T b ) {
        if ( a.compareTo( b ) > 0 ) {
            return a;
        } else {
            return b;
        }
    }
}
```

1) Ici, la classe **Utils** n'a aucun paramètre de type, 2) **max** est une méthode de classe, 3) **max** a un paramètre de type, 4) la déclaration de type est locale à la méthode **max**.

Utilisation des méthodes génériques

L'utilisation d'une méthode générique ne demande (en général) aucun élément syntaxique supplémentaire. L'inférence de l'argument de type est automatique.

```
Integer i1, i2, iMax;

i1 = new Integer( 1 );
i2 = new Integer( 10 );

iMax = max( i1, i2 );

System.out.println( "iMax = " + iMax );
```

Ici, fait l'inférence que le type est **Integer**.

Utiliser une méthode générique

```
String s1, s2, sMax;

s1 = new String( "alpha" );
s2 = new String( "bravo" );

sMax = max( s1, s2 );

System.out.println( "sMax = " + sMax );
```

Ici, le compilateur fait l'inférénce que le type est **String**.

Utiliser une méthode générique

On peut toute fois fournir explicitement les valeurs de type à l'aide de la construction syntaxique suivante.

```
iMax = Utils.<Integer>max( i1, i2 );
sMax = Utils.<String>max( s1, s2 );
```

C'est parfois nécessaire, lorsque le contexte est ambigu . . .

Type de données

Tel que discuté au premier cours, un type de données, est caractérisé par,
– un ensemble de valeurs ;
– un ensemble d'opérations ;
– une représentation.
Ces caractéristiques sont nécessaires au moment de la compilation afin de vérifier la validité d'un programme.

Et aussi, afin d'allouer l'espace mémoire nécessaire.

Type Abstrait de Données – TAD

(**abstract data type — ADT**)

– Ensemble de valeurs ;
– Ensemble d'opérations.

c.-à-d. aucune représentation !

Un type de données concret doit spécifier la représentation des données.

Un TAD permet de séparer clairement ≪l'utilisation≫ de ≪l'implémentation≫ (qui est privée).

Un type abstrait de données défini les opérations valides sans en définir l'implémentation.

Les interfaces permettent donc la spécification d'un type abstrait en Java.

```
public interface Stack {
     public abstract boolean isEmpty();
     public abstract Object peek();
     public abstract Object pop();
     public abstract Object push( Object item );
}


public interface Queue {
     public abstract boolean isEmpty();
     public abstract Object dequeue();
     public abstract Object enqueue( Object item );
}
```

```java
public interface List {
    public abstract void add( int index, Object o );
    public abstract boolean add( Object o );
    public abstract boolean contains( Object o );
    public abstract Object get( int index );
    public abstract int indexOf( Object o );
    public abstract boolean isEmpty();
    public abstract Object remove( int i );
    public abstract int size();
}
```