ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of January 26, 2013

Abstract

- Interface
- Abstract data types

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Interface 1

In the general context of object-oriented programming (including but not restricted to Java), the interface designates the list of public methods and variables of a class.

For example, consider a class **Time** having three public methods, **getHours**, **getMinutes** and **getSeconds**, and no other public methods or variables, then, the interface of the class **Time** is **getHours**, **getMinutes** and **getSeconds**.

Problem 1

Through a series of problems, we will discover the need for a new concept, called **interface**.

Problem 1: write a (polymorphic) method that sorts an array of objects.

There is a variety of sort algorithms, including **bubble sort**, **selection sort** and **quick sort**.

What do all these algorithms have in common? (do not answer now)

To make this example concise, the size of the input array will be two (2).

Therefore, sorting the array will be simple: if the value of the first element is smaller than the value of the second element, there is nothing to do, otherwise exchange the content of the two cells.

An algorithm to solve the general case will also be presented.

Specifically, here is an implementation that sorts an array of two integers.

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

$\Rightarrow$ What are the necessary changes so that the method can be used to sort an array of **Time** objects?

1) Changing the type of the variables (parameter and local variable) as well as 2) replacing the comparison operator by a method call.

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

What would be the necessary changes so that the method can be used to sort an array of Shapes?

1) Changing the type of the variables (parameter and local variable) and replacing the method that is used to compare two values.

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

What do these algorithms have in common?

To sort an array of objects, one simply needs a way to compare two objects.

Sorting an array of objects is a task that is likely to occur in a variety of contexts, an array of student objects, bank accounts, transactions, etc. therefore a general, polymorphic, method would be useful.

What are the requirements/types/operations?

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

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

      __________ tmp;

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

1. Needs a method for comparing two objects;

2. The name of the method must be the same for all the classes who need to use the sort method;

3. The particular implementation depends on the type of the objects that are stored in the array.

Who am I?

Solution 1: super-class Comparable

This solution will be discarded later.

However, we are first trying to solve this problem with the tools that we have: object-oriented programming and inheritance.

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

What is missing?

Element 1 is the keyword “abstract”. Indeed, the implementation of the method compareTo depends on the particular type of object.

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

What is missing?

Finding the second element is slightly more complex, let's consider the implementation of the method **sort2** first.

```
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;
     }
  }
}
```

If your answer was **Comparable**. It was a good answer.

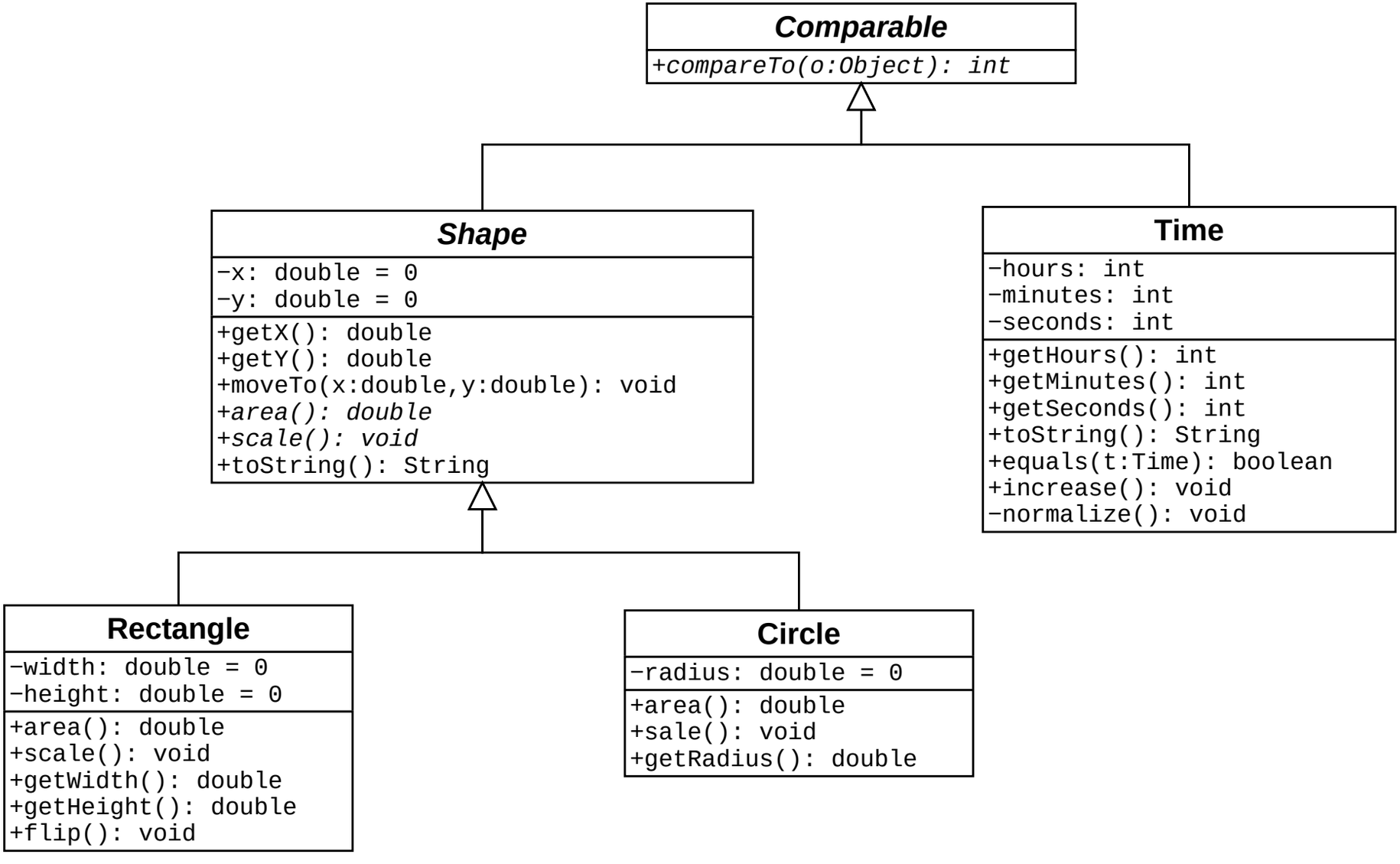
However, using **Object** will make the method slightly more general.

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

Our first solution requires creating an abstract class called **Comparable** that contains only one method, which is abstract, the method is called **compareTo**.

Any class who needs to use the (polymorphic) **sort2** method must be a sub-class of the class **Comparable** (and therefore must implement the method **compareTo**).

For instance, the classes **Time** and **Shape** would be modified so that both would be subclasses of **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

For the class **Time**, one would write,

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

      int result;
      if ( timeInseconds < other.timeInSeconds ) {
         result = -1;
      } else if ( timeInSeconds == other.timeInSeconds ) {
         result = 0;
      } else {
         result = 1;
      }
      return result;
  }
}
```

Hum, there is a compile-time error. What is it?

The problem is caused by the following signature **public int compareTo(Object other)**.

What is the problem?

What would be the error message?

```
‘‘Time is not abstract and does not override abstract method
  compareTo(java.lang.Object) in Comparable’’
```

For the class Time.

```
public class Time extends Comparable {
  ...
  public int compareTo( Object other ) {
      int result;

      if ( timeInseconds < other.timeInSeconds ) {
         result = -1;
      } else if ( timeInSeconds == other.timeInSeconds ) {
         result = 0;
      } else {
         result = 1;
      }
      return result;
  }
}
```

Warning! It would be tempting to write "public int compareTo(Time obj)", you must resist!

Grrr, the compilation of this class still produces a compile-time error, which one? Solution?

```
public class Time extends Comparable {
  ...
  public int compareTo( Object other ) {
      int result;

      if ( timeInseconds < other.timeInSeconds ) {
         result = -1;
      } else if ( timeInSeconds == other.timeInSeconds ) {
         result = 0;
      } else {
         result = 1;
      }
      return result;
  }
}
```

The parameter is of type **Object**, the instance variable **timeInSeconds** is not defined in the class **Object**. Solution:

```
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;
  }
}
```

Similarly, the class **Shape** needs to be changed so that 1) it becomes a subclass of **Comparable** and 2) it implements the method **compareTo** so that the method **Array.sort2** can be used to sort arrays of Shapes.

```
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;
    }
}
```

Here is a method that can be used to sort any array of (2) objects as long as the class of the objects is a subclass of **Comparable**, i.e. they have an implementation for **compareTo**.

```
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 );
```

Problem solved! For now . . .

Problem 2

Problem 2: write a (polymorphic) method that displays all the elements of an array (arrays of shapes, buttons, balloons, etc.).

```
class Graphics {

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

What should the type of the elements of this array?

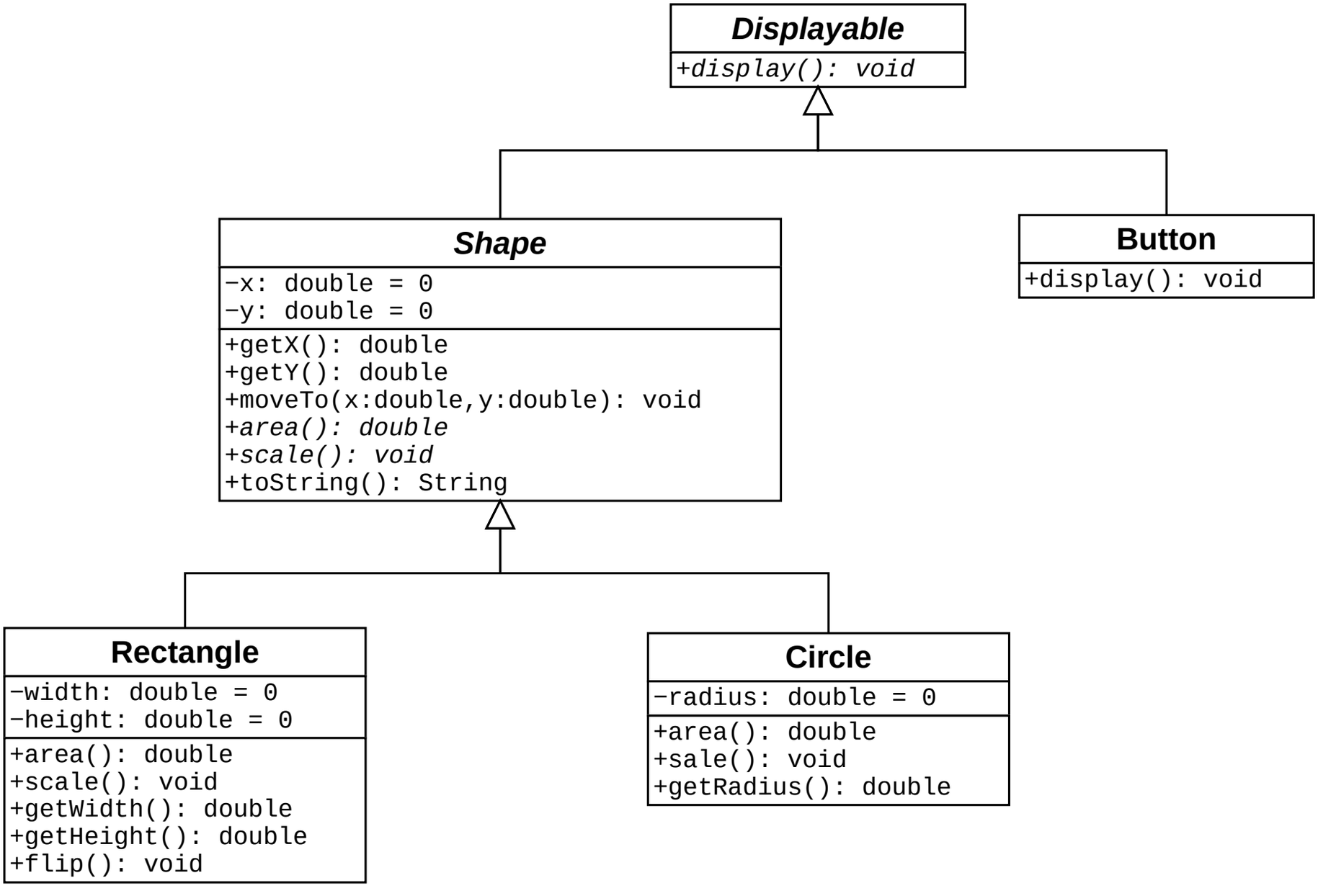
```
public _________ class Displayable {
    public _________ void display();

}
```

What is missing?

```
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

Usage: Any class who needs to use the method **Graphics.displayAll(Displayable[] as)** must:

1. Be a subclass of **Displayable**;

2. Implement the method display.

Problem solved! For now . . .

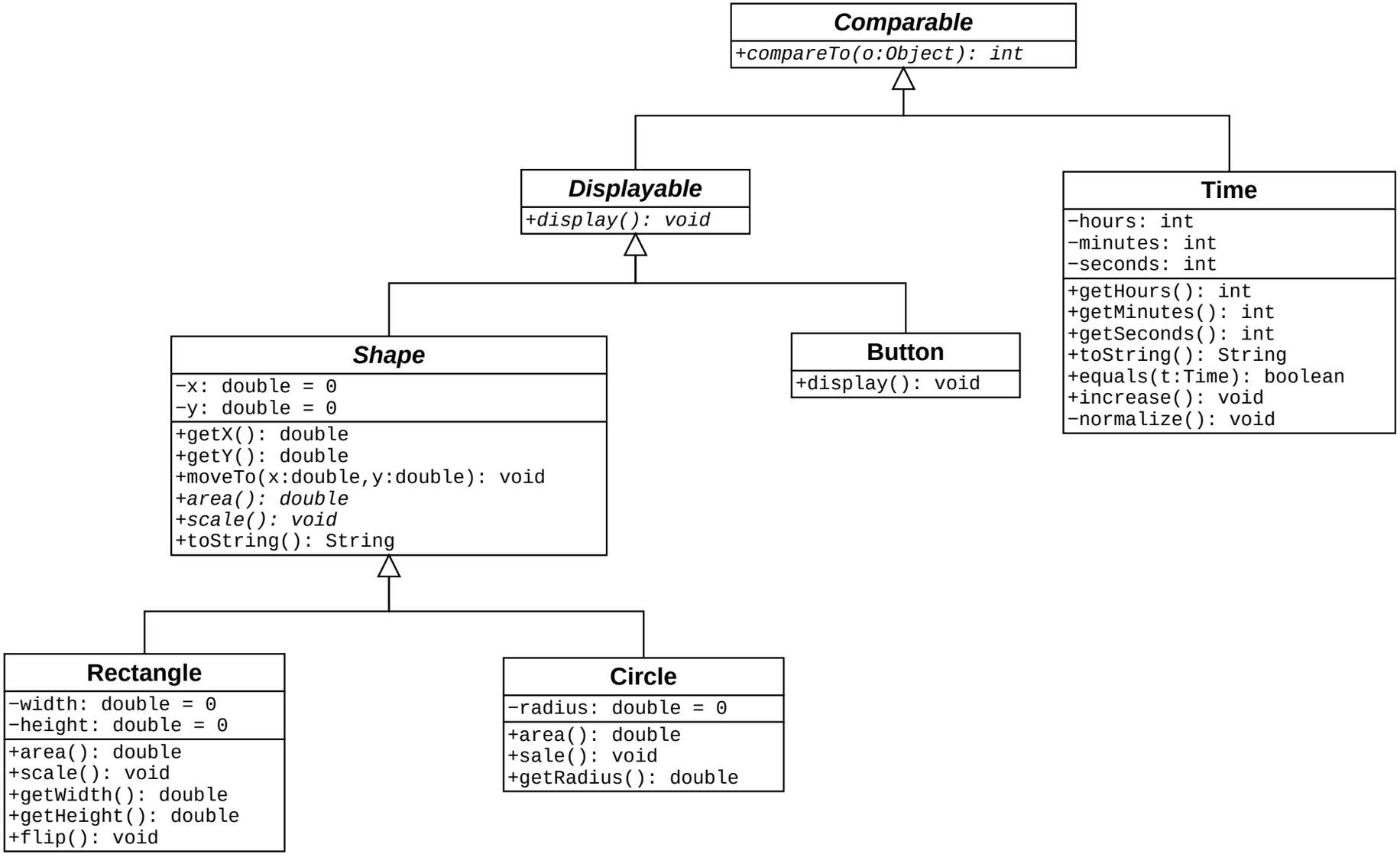
Problem 3

Problem 3: Shape should be both **Comparable and Displayable**!

What solution do you propose?

One possible solution would be to make **Displayable** a subclass of **Comparable** therefore forcing any of its subclasses to implement both **display()** and **compareTo()**.

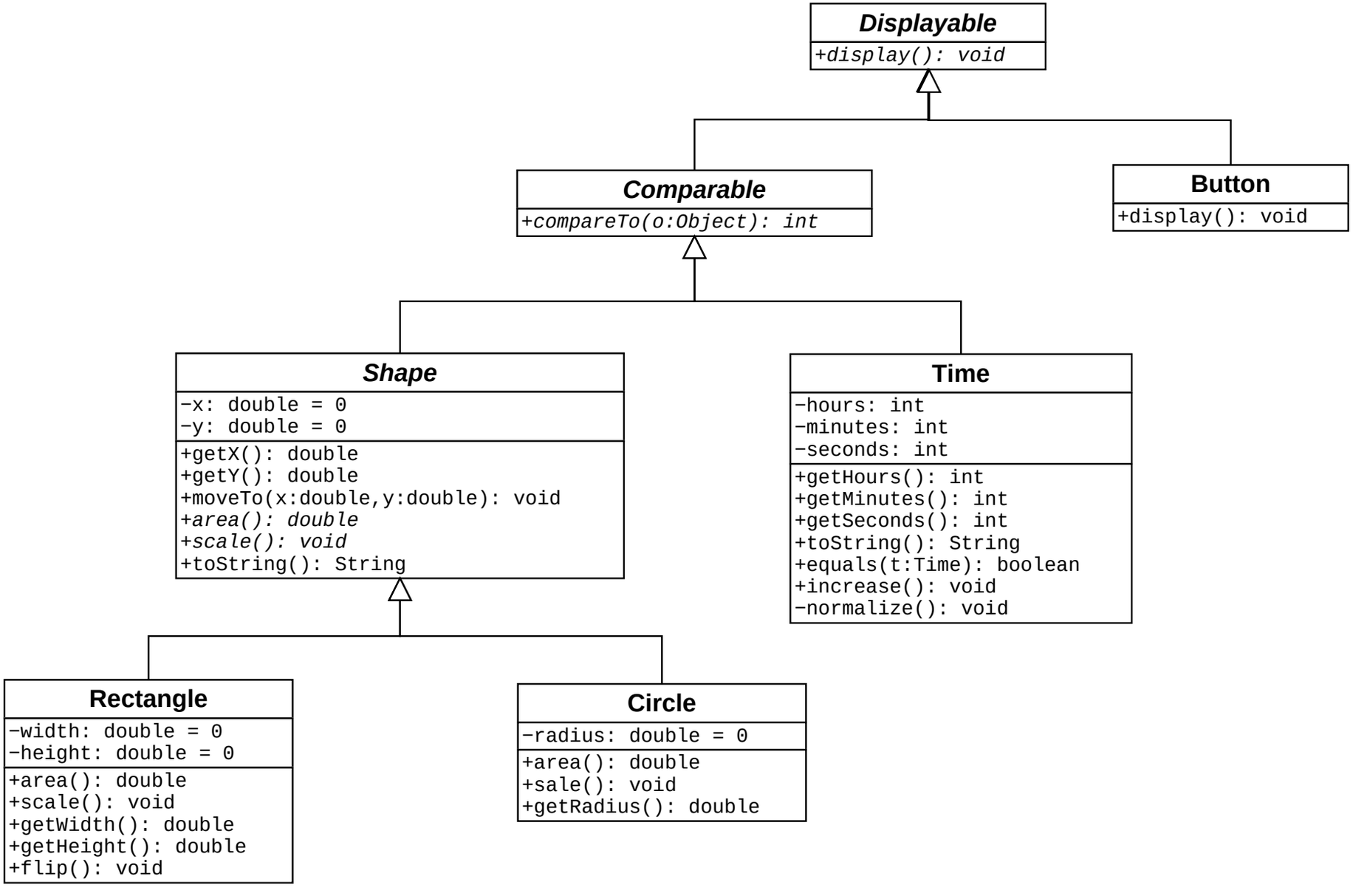
What do you think?


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

The problem with this hierarchy is that **Button** also has to be **Comparable** and **Displayable**.

Maybe it does make much sens for a button to be **Comparable**!

What do we do, let's change the hierarchy.

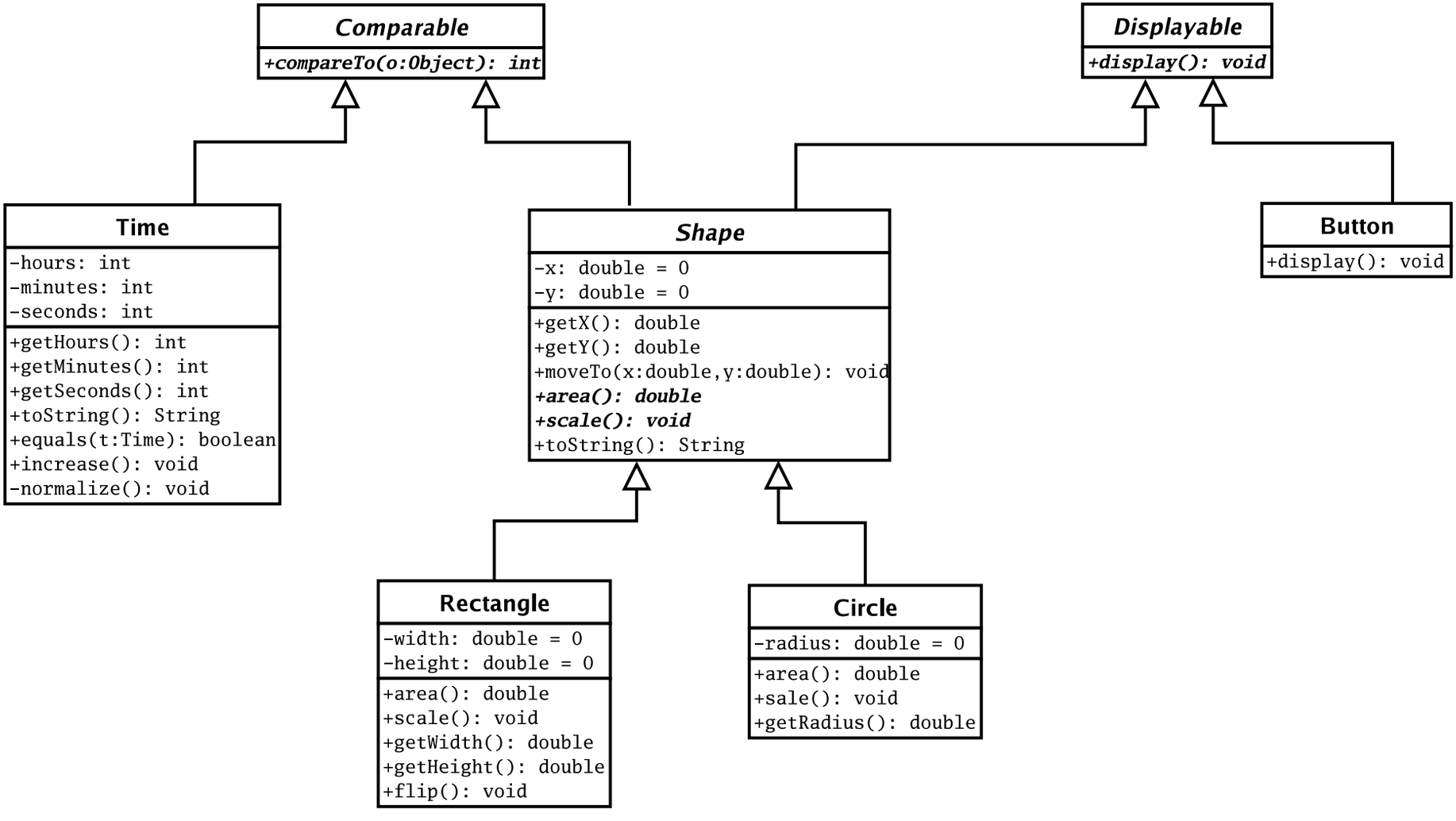

Displayable
+display(): void
Comparable
+compareTo(o:Object): int
Button
+display(): void
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): double
+moveTo(x:double,y:double): void
+area(): double
+scale(): void
+toString(): String
Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+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

The problem with this new hierarchy is that **Time** is now **Displayable** as well as being **Comparable**.

It doesn't make much sens to for the class **Time** to implement the method **display()**.

It's not possible to organize these classes coherently using single inheritance.

It seems what we need is multiple inheritance?

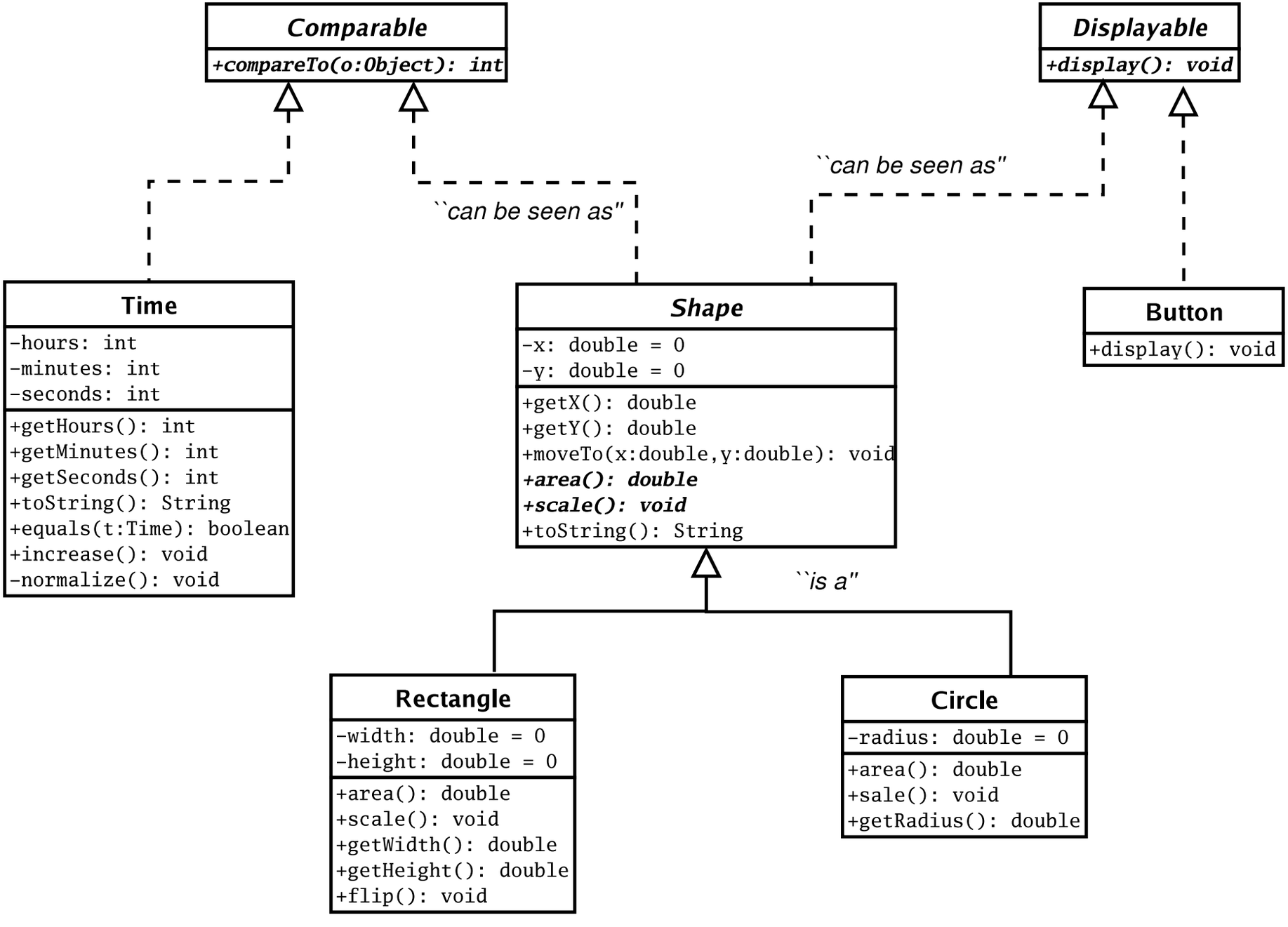

Comparable
+compareTo(o:Object): int
Displayable
+display(): void
Time
-hours: int
-minutes: int
-seconds: int
+getHours(): int
+getMinutes(): int
+getSeconds(): int
+toString(): String
+equals(t:Time): boolean
+increase(): void
-normalize(): void
Shape
-x: double = 0
-y: double = 0
+getX(): double
+getY(): 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

The problem is that Java does not support multiple-inheritance.

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

Dead-end.

Java has an alternative concept to solve particularly this kind of problem, it's called an **interface**, and it implements the relationship "can be seen as" (as opposed to "is a", that class inheritance implements).


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

An interface definition resembles a class definition.

It consists of the keyword **interface**, instead of **class**, followed by the name of the interface.

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

The definition is put in a file that has the same name as the interface and a .java extension.

The file is compiled, as a class would be, to produce a **.class** file.

Interface

An interface contains:

- Constants;
- Abstract methods definitions.

Like an abstract class, it's not possible to create an instance of an interface.

Unlike an abstract class the interface cannot contain concrete methods.

Comparable

The interface **Comparable** is actually part of the standard Java library!

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

> "This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.
>
> The method compareTo Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object."

We have defined an abstract data type (Comparable), it needs an implementation.

```
public abstract class Shape implements Comparable {


}
```

New keyword, **implements**. The class **Shape** implements the interface **Comparable**.

So!? The class **Shape** must provide an implementation for the method **int compareTo(Object o)**.

A class that implements an interface must implement all the methods listed in the interface.

What have we gained? A class can implement more than one interface.

```
public abstract class Shape implements Comparable, Displayable {

}
```

It simply means that **Shape** must implement both methods **compareTo** and **display**.

What is it useful for?

```
Comparable a;  // valid?
Comparable b;  // valid?

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

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

An interface can be used as the type of a reference variable.

```
Comparable o;
```

What can you store in it?

Is this a valid statement?

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

No. The following statement is not valid.

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

Those two statements are valid!

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

The reference of an object that implements the interface can be stored in a reference of this type.

```
o.getX(); // not valid
c.getX(); // valid
```

Pitfall, are those statements valid?

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

A reference always designates an object and all the objects have the characteristics of the class **Object**.

Usage

The class **String** implements the interface **Comparable**.

The standard library has sort method that sorts arrays of **Comparable** objects.

```
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] );
    }
}
```

```
> java Test using interfaces in Java

Java
in
interfaces
using
```

Similarly, if the **Time** class definition is modified so that it implements the interface **Comparable**, i.e. its declaration is modified as follows:

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

and accordingly it must implement the method **compareTo**, then the same sorting algorithm can be used to sort a **Time** array.

```
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;
            // Find the smallest element in the unsorted
            // region of the array.
            for ( int j = i+1; j < a.length; j++ )
                if ( a[ j ].compareTo( a[ min ] ) < 0 )
                    min = j;
            // Swap the smallest unsorted element with
            // the element found a position i.
            Comparable tmp = a[ min ];
            a[ min ] = a[ i ];
            a[ i ] = tmp;
        }
    }
}
```

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

Implementing Multiple Interfaces

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

$\Rightarrow$ Java does not allow multiple inheritance but it does allow a class to implement several interfaces.

The interface resembles an abstract class.

However, there are only abstract methods in an interface.

An abstract class may also have concrete methods.

Interfaces and the type system

An interface is a conceptual tool for the programmer to design software.

What does it explicitly mean to the compiler?

Let's consider, the interface **Comparable**, the class **SortAlgorithms**, which declares variables of type **Comparable**, and the class **Shape**, which implements **Comparable**.

These classes represent the three players for these equations, the interface, an implementation and a polymorphic method.

A interface defines a contract. (**Comparable**)

A class that declares variables of an interface type has to use those variables consistently w.r.t. the declaration of the interface, i.e. it can only use the methods that are declared in the interface. (**SortAlgorithms**)

A class that "implements" an interface has to provide an implementation for every method listed in the interface. (**Shape**)

A class that declares variables of an interface type, such as **SortAlgorithms**, can be compiled in the absence of an actual implementation, specifically to compile **SortAlgorithms** all that's needed is **SortAlgorithms.java** and the interface (here, this interface exists in Java's own library), an actual implementation for the interface is not required.

Interfaces vs Abstract Classes

Interfaces and abstract classes are two ways to define a data type with an abstract contract (i.e. without implementation).

Which one to use?

- An abstract class that contains only abstract methods should probably be defined as an interface;
- If the problem needs multiple-inheritance then use interfaces;
- To mix concrete and abstract methods you need to use an abstract class;
- An interface defines the relationship "can be seen as";
- Inheritance defines the relationship "is a".

Interfaces vs Abstract Classes

Property	**Concrete**	**Abstract**	**interface**
Instances can be created	Yes	No	No
Instance variables and methods	Yes	Yes	No
Constants	Yes	Yes	Yes
Can declare abstract methods	No	Yes	Yes

Interfaces

An interface is useful when there are several possible implementations for a given problem/data structure.

Take the example of the **Time** class, defining an interface that contains all the necessary an sufficient methods for the time concept would allow us to create programs that would work with either of the two implementations.

An abstract data type defines the valid operations without providing an implementation.

An interface is therefore a formalism that can be used to express abstract data types in Java.

Generic and Parameterized Types

Defining a generic type.

A **generic type** is a **reference type** that has one or more type parameters.

A **generic type** is an **interface** or **class** that has one or more type parameters.

Defining a generic type

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

Creating a parameterized type

```
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;
    }

}
```

Generic methods

Besides generic types (classes with formal type parameters), methods can also have type parameters.

Generic methods

```
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) Herein, the class **Utils** has no type parameter, 2) **max** is a class method, 3) **max** has a type parameter, 4) the type declaration is local to the method **max**.

Using generic methods

Invoking a generic method requires no additional syntactical construction. The type argument is inferred automatically.

```
Integer i1, i2, iMax;

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

iMax = max( i1, i2 );

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

Here, the compiler infers the type argument **Integer**.

Using generic methods

```
String s1, s2, sMax;

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

sMax = max( s1, s2 );

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

Here, the compiler infers the type argument **String**.

Using generic methods

The type may also be supplied explicitely:

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

this is necessary whenever the type system cannot infer a type that is specific enough.

Data Types

As discussed in the first lecture, a **data type** is characterized by:

- a set of values;
- a set of operations;
- a data representation.

These characteristics are necessary for the compiler to verify the validity of a program — which operations are valid for a given data.

These characteristics are also necessary for the compiler to be able to create a representation for the data in memory; how much memory to allocate for example.

Abstract Data Type

An **abstract data type (ADT)** is characterized by:

- a set of values;
- a set of operations.

i.e. the data representation is not part of specification of an ADT.

A concrete data type must have a representation, but the ADT makes a distinction between "how it is used" and "how it is implemented", which is private.

Abstract Data Type (contd)

The design of the **Time** classes followed this principle.

Both implementations can be characterised by the following behaviour:

- Allow represent a time value with a precision of one second for a period of 24 hours.
- Both classes have the list of arguments for their respective constructor.
- Both classes implement: getHours(), getMinutes(), getSeconds(), increase (), before(t), after(t) and equals(t), where **t** is an instance of a time class.

ADT specification in Java

The ADT concept is independent of any programming language. It's a discipline to avoid tight coupling of the classes.

In Java, whenever a class is created that has no public variables, such as Time1 and Time2, it expresses and ADT.

This idea is so important that Java has a specific concept to express ADTs, an interface. An interface is a pure abstract contract.

```
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 );
}
```

```
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();
}
```