ITI 1120
Lab #9

Slides by: Diana Inkpen, Alan Williams, Daniel Amyot

Some original material by Romelia Plesa

Objectives

- Review fundamental concepts
- Example: the Time class
- Exercises
  - Modify the Time class
  - The Car class

Some Concepts

- Classes
  - A class contains variables and methods (code)
  - Think of a class as a **type** of complex variable that provides a template to create objects.
  - The object contains complex data (many variables of different types) and the possible operations on these variables (the methods).
- Reference to an object
  - The reference variable contains the reference (an address) to an object. The following is a declaration of a reference variable.

```
ClassName refVariableName;
```

  - Recall that after a reference variable is declared, it contains a null value (does not reference an object).

Some Concepts (continued)

- Creating an object
  - An object MUST BE created to be used:

```
refVariableName = new ClassName( arg list );
```

  - The method after `new` is the class constructor – its function is to initialise any variables of the new object.
  - Instance variables: variables defined in the class and created within the object.
  - Instance methods: the code that offers operations with the object variables.

A "Time" class

- Suppose we want to be able to work with values representing clock times to 1 minute precision.
  - What information to we have to store to represent a time? How do we store that information?
  - What operations might we want to do with a Time value?

What to store in "Time"?

- Two integers:
  - **hour**: the hour number ($0 \leq$ **hour** ≤ 23)
  - **minute**: the minute number ($0 \leq$ **minute** ≤ 59)

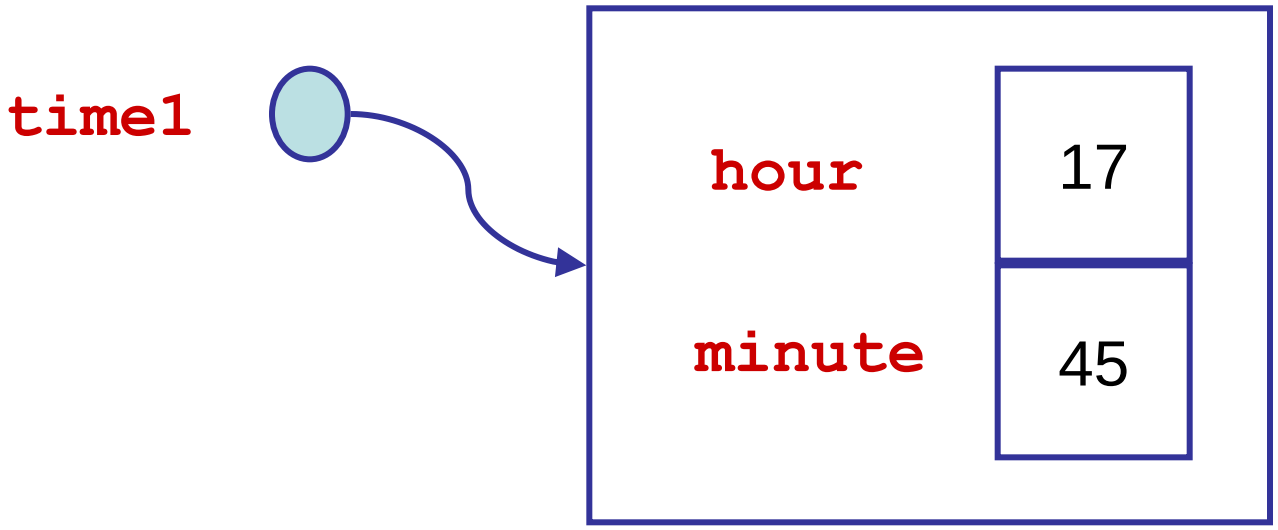
```
class Time
{
    public int hour;
    public int minute;
}
```

- Alternatives?
  - Only minutes or seconds from midnight!

Declaring and creating Time objects

```
// DECLARE VARIABLES / DATA DICTIONARY
Time time1;                          // time is null here
…
// ALGORITHM BODY
…
time1 = new Time( );  // We now have a Time object
                      // but it contains ? for values
time1.hour = 17;
time1.minute = 45;    // time1 now represents 17:45
```

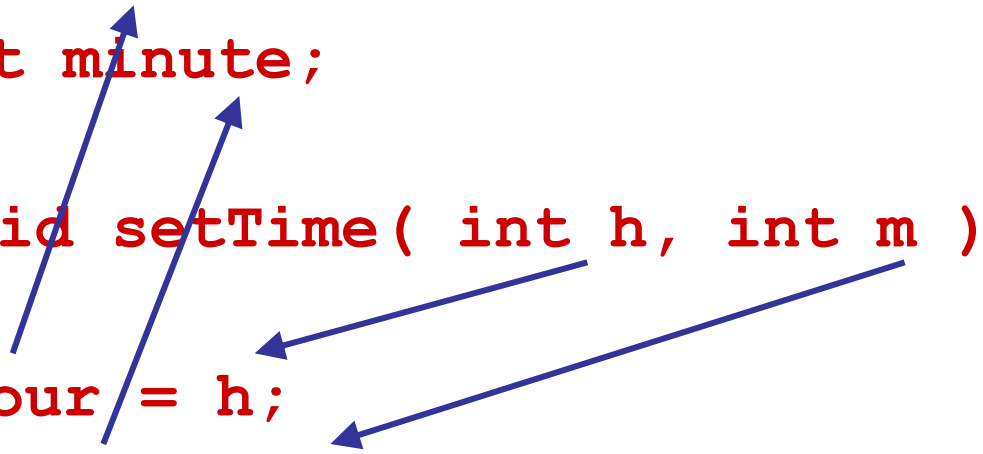
What do we have?



A method to set the time

```
class Time
{
    public int hour;
    public int minute;

    public void setTime( int h, int m )
    {
        this.hour = h;
        this.minute = m;
    }
}
```



Usage

```
// DECLARE VARIABLES / DATA DICTIONARY
Time time1;                          // time is null here
…
// ALGORITHM BODY
…
time1 = new Time( );
time1.setTime( 17, 45 );

// time1 now represents 17:45
```

This method is different!

- Did anyone see what was "missing" from the method:

```
public void setTime( int h, int m )
{
    this.hour = h;
    this.minute = m;
}
```

- The word **static** does not appear in the method header.

Instance methods

- When the word **`static`** does not appear in the method header, this means that the method can be called via a variable declared to be of a type matching the class name. This is called an "instance" method.

- An instance method can make use of the variables defined in the class.

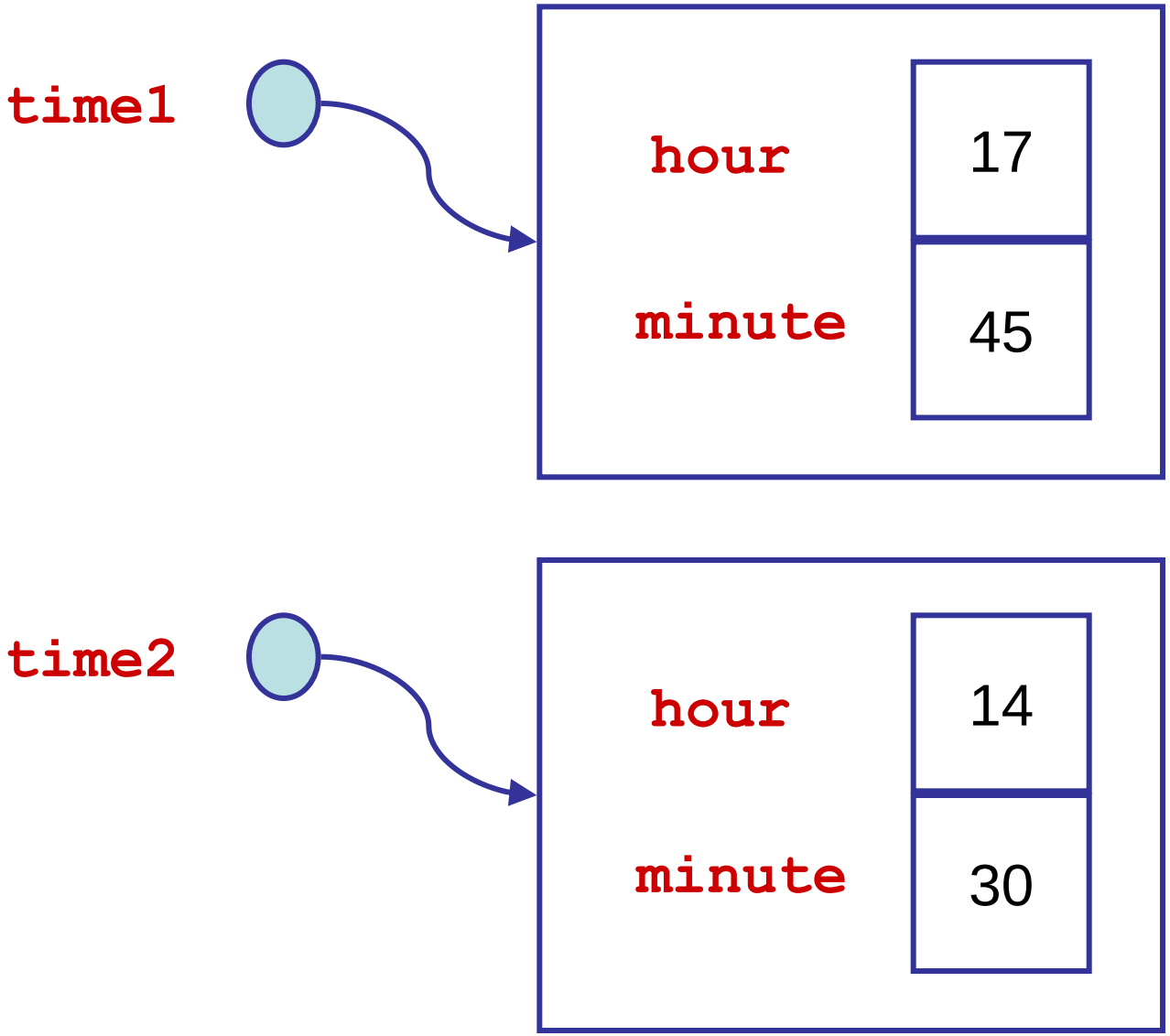
- The result: the method will produce different results for different object instances.

Instance methods

- For example,

```
Time time1;
Time time2;
time1 = new Time( );
time2 = new Time( )
time1.setTime( 17, 45 ); // time1 is 17:45
time2.setTime( 14, 30 ); // time2 is 14:30
```

What do we have?



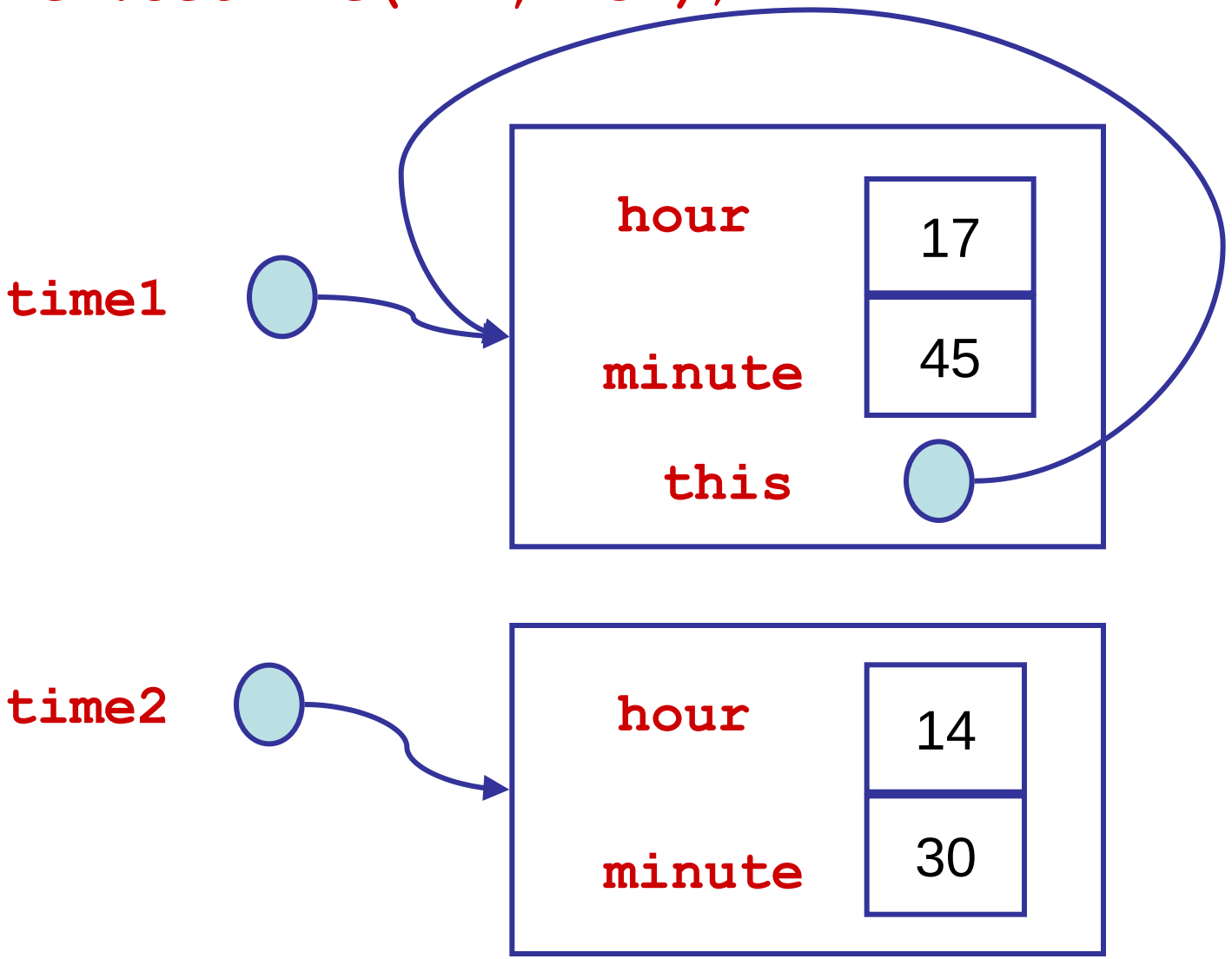
this

- Objects **time1** and **time2** use the same code in class **Time** to set their own copy of **hour** and **minute**.

- When we want to refer to "the object on which I was called", we use **this**.

this

```
time1.setTime( 17, 45 );
```



Information Hiding

- If we want to ensure that:
  - **hour**: must be in range $0 \leq$ **hour** ≤ 23
  - **minute**: must be in range $0 \leq$ **minute** ≤ 59

  then direct access to these variables should not be permitted.

```
class Time
{
    private int hour;
    private int minute;
}
```

- Instead, access should be provided through **setTime**, and we can adjust the values if needed.

Revised version of **setTime**

```
public void setTime( int h, int m )
{
   // If minutes value is too large, adjust it
   // by mod 60, and add to hours value.

   if ( m > 59 )
   {
      h = h + m / 60;   // determine hours to add
      m = m % 60;       // puts minutes in range
   }
   else
   {
      ; // do nothing
   }
   this.hour = h % 24;  // puts hours in range
   this.minute = m;
}
```

Accessors

- With the variables now declared to be **private**, we need to provide a way for other classes to ask for the values.

```
public int getHours()
{
   return hour;
}
public int getMinute( )
{
   return minute;
}
```

Compare times for equality

- Suppose we want a method that checks whether one time object is equal to another.

- One approach: a **`static`** method

```
public static boolean isEqual( Time t1, Time t2 )
```

  This would be called as **`Time.isEqual( t1, t2 )`**

- Alternative: an instance method

```
public boolean isEqual( Time t2 )
```

  This would be called as **`t1.isEqual( t2 )`**

The `static` method

```
public static boolean isEqual( Time t1, Time t2 )
{
   return (t1.hour == t2.hour) &&
          (t1.minute == t2.minute);
}
```

- If the method is inside the class **`Time`**, it can access the private variables inside the class.
- Why are there 2 parameters in the above case?

The instance method

```
public boolean isEqual( Time t2 )
{
   return (this.hour == t2.hour) &&
          (this.minute == t2.minute);
}
```

- In this case, we are comparing "ourself" to another **`Time`** value.
- Why is only one parameter sufficient in this case?

Exercise 1

- Add the following methods to the **Time** class:

```
public boolean isBefore( Time t2 )
```

Returns **true** if the time represented by **this** is before the time in **t2**, and false otherwise

```
public Time duration( Time t2 )
```

Returns a new **Time** object with the number of hours and minutes between **this** and **t2**.

- Write a **main** method in the class TestTime to test your new methods (or create JUnit tests).

- If you have time, add a display method:

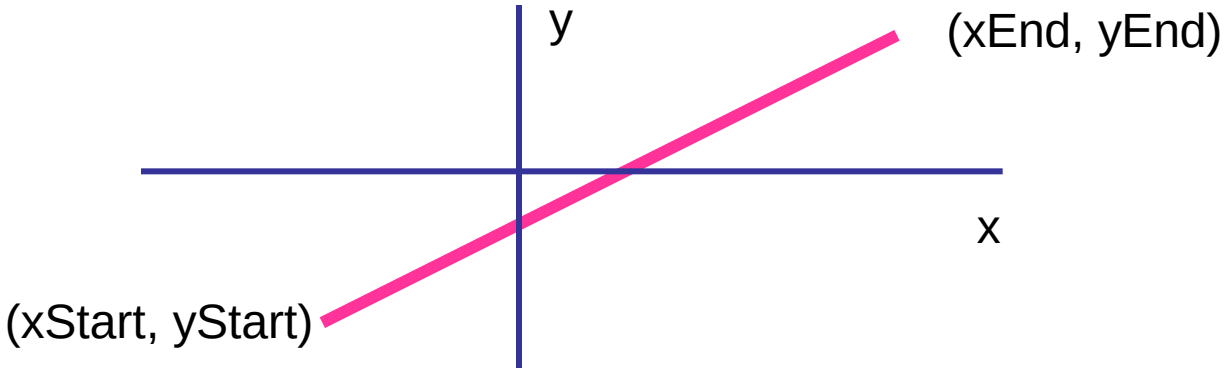
```
public String toString()
```

which allows the display of the Time object according to the format hour:minute in a print/println (for example `System.out.println(time1);`)

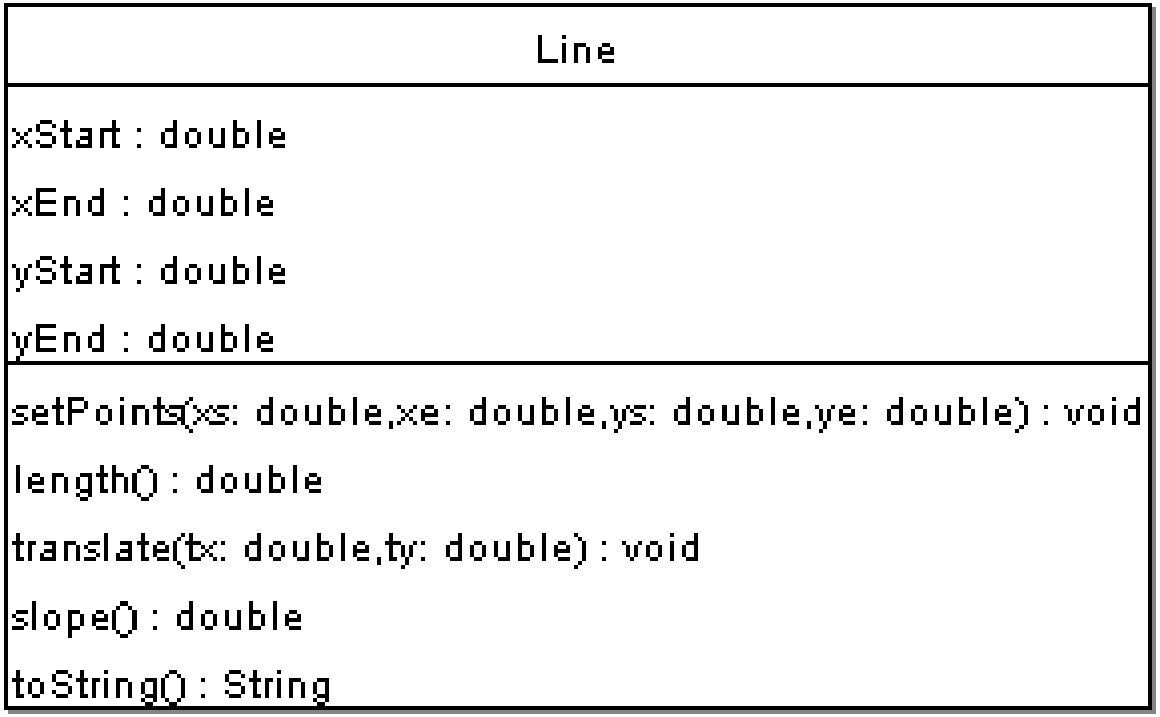
A "Line" class

- Design a Java class **`Line`** that will store information for a line, where the line is in an (x,y) coordinate space, and provide operations to work with lines.

- Each line object starts at a point (xStart, yStart) and ends at a point (xEnd, yEnd), where xStart, yStart , xEnd, and yEnd are real-valued numbers that could be positive or negative.



UML diagram for **Line**



Method descriptions

- Set the start and end points of the line.
  - Name of method: **`setPoints(...)`**
  - Parameters to the method: **`xs`**, **`ys`**, **`xe`**, **`ye`**
  - Results: (none)
  - Modified: the line object
-
- Return the length of the line
  - The length is $\sqrt{(y_e - y_s)^2 + (x_e - x_s)^2}$
    - Name of method: **`length( )`**
    - Parameters to the method: none.
    - Result: length of the line (a real value)

Method descriptions

- Translate the line by (tx, ty), where tx and ty are any positive or negative real values.
  - A *translation* of a line represents "sliding" the entire line. The value tx is added to the x coordinates of the start and end of the line, and the value ty is added to the y coordinates of the start and end of the line.
    - Name of method: **`translate(...)`**
    - Parameters to the method: **`tx`**, **`ty`**
    - Results: (none)
    - Modified: the line object

Method descriptions

- Return the slope of the line.
  - The slope is $(ye - ys) / (xe - xs)$ (watch out for vertical lines – their slope is infinite!)
    - Name of method: `slope( )`
    - Parameters to the method: none.
    - Result: the slope of the line (a real value)

Method descriptions

- Returns a `String` with information about the line.
  - The string that is returned for a line with (for example) start point (0.0,1.0) and end point (3.5,-1.2) should be:

    ```
    Line from (0.0, 1.0) to (3.5, -1.2)
    ```

  - Formatting of the values to a specific number of decimal places is not required.
    - Name of method: `toString ( )`
    - Parameters to the method: (none)
    - Results: a `String` in the above format

Exercise

- Implement the class **Line**
  - **implement the toString method**
  - **add a printLineInfo method**

- Test the class using the main method in **LineTest.java**