Chulalongkorn University
International School of Engineering
Department of Computer Engineering
2140105 Computer Programming Lab.

Name ______________________
Student ID __________________
Station No. __________________
Date ________________________

Lab 9 – OO Concept (Episode III)

Objectives:

- Understand the concept of interface
- Understand interfaces in Java
- Be able to define a class that implements an interface.

Interfaces [from Sun's Java Tutorial]

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, makes computer systems that receive GPS (Global Positioning Satellite) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know how the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

Interfaces in Java

In the Java programming language, an *interface* is a *reference type*, similar to a class, which can contain only constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {
    // constant declarations, if any

    // method signatures
    int turn(Direction direction,   // An enum with values RIGHT, LEFT
             double radius, double startSpeed, double endSpeed);
    int changeLanes(Direction direction, double startSpeed, double endSpeed);
    int signalTurn(Direction direction, boolean signalOn);
    int getRadarFront(double distanceToCar, double speedOfCar);
    int getRadarRear(double distanceToCar, double speedOfCar);
     ......
    // more method signatures
}
```

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

```
public class OperateBMW760i implements OperateCar {

   // the OperateCar method signatures, with implementation --
   // for example:
   int signalTurn(Direction direction, boolean signalOn) {
      //code to turn BMW's LEFT turn indicator lights on
      //code to turn BMW's LEFT turn indicator lights off
      //code to turn BMW's RIGHT turn indicator lights on
      //code to turn BMW's RIGHT turn indicator lights off
   }

   // other members, as needed -- for example, helper classes
   // not visible to clients of the interface

}
```

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

Interfaces and Multiple Inheritance

Interfaces have another very important role in the Java programming language. Interfaces are not part of the class hierarchy, although they work in combination with classes. The Java programming language does not permit multiple inheritance, but interfaces provide an alternative.

In Java, a class can inherit from only one class but it can implement more than one interface. Therefore, **objects can have multiple types**: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface. This is discussed later in this lesson, in the section titled "Using an Interface as a Type."

Defining an Interface

An interface declaration consists of modifiers, the keyword `interface`, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

   // constant declarations
   double E = 2.718282;  // base of natural logarithms

   // method signatures
   void doSomething (int i, double x);
   int doSomethingElse(String s);
}
```

The `public` access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is `public`, your interface will be accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

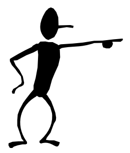
The Interface Body

The interface body contains method declarations for all the methods included in the interface. A method declaration within an interface is followed by a semicolon, but no braces, because an interface does not provide implementations for the methods declared within it. All methods declared in an interface are implicitly `public`, so the public modifier can be omitted.

An interface can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly `public`, `static`, and `final`. Once again, these modifiers can be omitted.

Implementing an Interface

To declare a class that implements an interface, you include an `implements` clause in the class declaration. Your class can implement more than one interface, so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class.



Your turn (1)

1. Create a new project called `lab9`.
2. Create a new package called `lab9interface`.
3. Create a new interface called `Relatable` in package `lab9interface` and copy the following code:

```
public interface Relatable {

   // this (object calling isLargerThan) and
   // other must be instances of the same class
   // returns 1, 0, -1 if this is greater
   // than, equal to, or less than other
   public int isLargerThan(Relatable other);

}
```

If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement `Relatable`.

Any class can implement `Relatable` if there is some way to compare the relative "*size*" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth. For planar geometric objects, area would be a good choice, while volume would work for three-dimensional geometric objects. All such classes can implement the `isLargerThan()` method.

If you know that a class implements `Relatable`, then you know that you can compare the size of the objects instantiated from that class.

4. Implementing the `Relatable` Interface by create a new class called `RectanglePlus` in package `lab9interface` and copy the following code:

```
public class RectanglePlus implements Relatable {
    public int width = 0;
    public int height = 0;
    public int x = 0;   // x-coordinate of upper left corner
    public int y = 0;   // y-coordinate of upper left corner

    public RectanglePlus(int x, int y, int w, int h) {
        this.x = x;
        this.y = y;
        this.width = w;
        this.height = h;
    }

    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }

    // a method to implement Relatable
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1;
        else
            return 0;
    }
}
```

Because `RectanglePlus` implements `Relatable`, the size of any two `RectanglePlus` objects can be compared.

5. Create a new application called `RectanglePlusDemo` in package `lab9interface`.
6. In the main method, instantiate two `RectanglePlus` objects with different size, and print out which is the larger `RectanglePlus` object.

Using an Interface as a Type

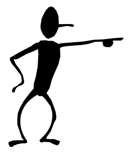
When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

As an example, here is a method for finding the largest object in a pair of objects, for any objects that are instantiated from a class that implements `Relatable`:

```
public Object findLargest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1; // casting to Relatable
   Relatable obj2 = (Relatable)object2; // casting to Relatable
   if ( (obj1).isLargerThan(obj2) > 0)
      return object1;
   else
      return object2;
}
```

By casting `object1` to a `Relatable` type, it can invoke the `isLargerThan` method.

This methods work for any "relatable" objects, no matter what their class inheritance is. When they implement `Relatable`, they can be of both their own class (or superclass) type and a `Relatable` type. This gives them some of the advantages of multiple inheritance, where they can have behavior from both a superclass and an interface.



Your turn (2)

Rewriting Interfaces

1. Create a new interface called `DoIt` in package `lab9interface`:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

2. Create a new class called `SomeClass` in package `lab9interface` that implements `DoIt`:

```
public class SomeClass implements DoIt {
   public void doSomething(int I, double x) { }
   public int doSomethingElse(String s) { }
}
```

3. Add a third method to `DoIt`, so that the interface now becomes:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);
}
```

If you make this change, all classes that implement the old `DoIt` interface will break because they don't implement the interface anymore. Programmers relying on this interface will protest loudly. You can see that, now `SomeClass` has an error indicated that it must implements inherited abstract method `DoIt.didItWork`.

Try to anticipate all uses for your interface and to specify it completely from the beginning. Given that this is often impossible, you may need to create more interfaces later.

4. Create a `DoItPlus` interface that extends `DoIt`:

```
public interface DoItPlus extends DoIt {
   boolean didItWork(int i, double x, String s);
}
```

5. Delete the third method you added in `DoIt`. The error in `SomeClass` is gone. Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

Summary of Interfaces

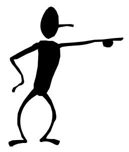
An interface defines a protocol of communication between two objects.

An interface declaration contains signatures, but no implementations, for a set of methods, and might also contain constant definitions.

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

An interface name can be used anywhere a type can be used.

Lab 9 Exercise



Your turn (3)

1. Run `lab9demo.jar` by double-click on the file or in Command Prompt, go to the folder where the file `lab9demo.jar` is. Type **`java –jar lab8demo.jar`**. You will see the demo program run as show in Figure 1. Try to click on the buttons an see the result. You need to make your program run like the demo program.

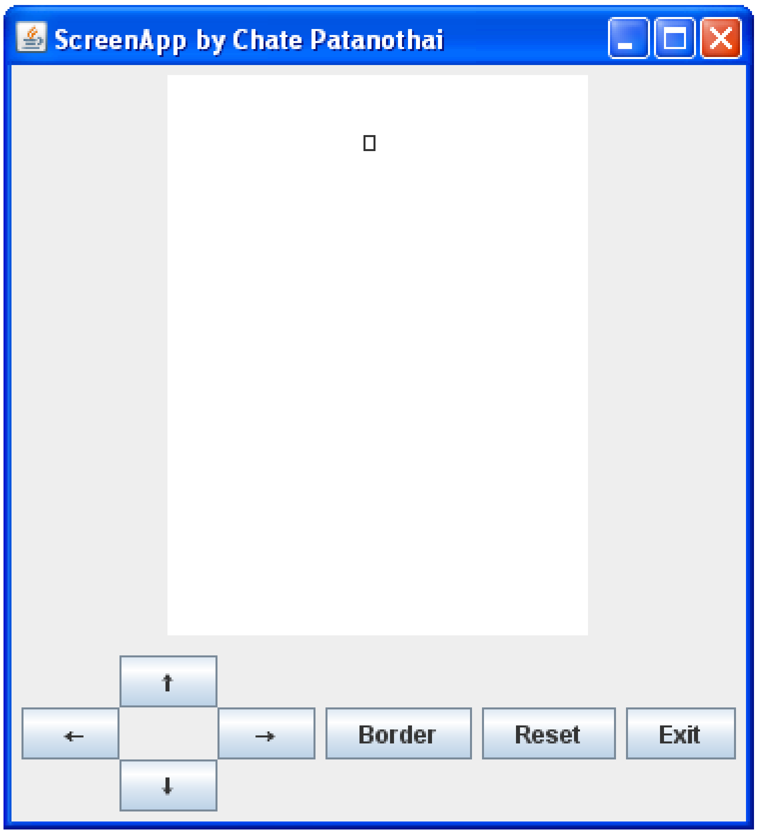


Figure 1 Demo program.

- Exit – quits the program
- Reset – clears everything and start the robot in a new random location in the map
- Border – draws the border of the map
- Right Arrow – moves robot one position to the right, if the current position is rightmost, the robot moves to leftmost.
- Left Arrow – moves robot one position to the left, if the current position is leftmost, the robot moves to rightmost.
- Up Arrow – moves robot one position up, if the current position is the top-most, the robot moves to the bottom of the map.
- Down Arrow – moves robot one position down, if the current position is the bottom-most, the robot moves to the top of the map.

2. Import **`lab9src.jar`** into your project. After that, you will have **`Map.java`**, **`Movable.java`**, and **`Robot.java`** in your default package.
   - Movable.java – the interface that define some methods
   - Robot.java – a class that implements Movable
   - Map.java – a class that represents the map which the robot moves in.
3. Start a command prompt window by click **Start > Run... > cmd.exe** and change directory to your project folder.
4. Type **`java ScreenApp`**, to run ScreenApp application. A window will popup as shown in Figure 2.

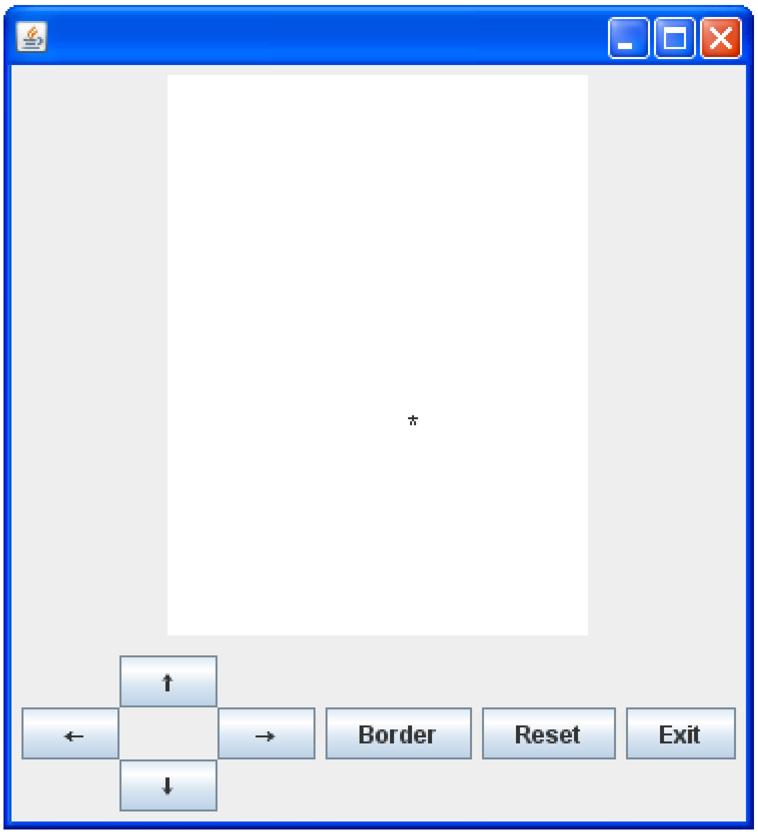


Figure 2 ScreenApp program.

5. Click on “Reset” button. You can see that a ‘*’ is displayed in random position every time you click this button. The method that does this job is method clear() in **`Map.java`**. To quit the program, click “Exit” button.
6. Look at the file Robot.java. The statement

   name = “”;

   set the title of this application. Change the name to your name, save Robot.java and run ScreenApp again. You can see that now the application has your name as its title.

7. When you click the “Border” button, the program executes method drawBorder() which is empty now. Add the statements that will make drawBorder() executes as described in the method’s comment or try the demo program.
8. Click on RightArrow button, you will see that the robot keep moving to the right until it reaches the rightmost, then moving to the leftmost. Look in the file Robot.java, the method moveRight() is the method that does the job. You can use this method as the example for implementing the remaining methods:
   - `moveUp()`, `moveDown()`, `moveLeft()`,for Up, Down, and Left Arrow button respectively.

   Your job is to add the code for those methods, so they behave as the specified in the comment or same as the demo program.

Name ______________________________
Student ID. _________________________
Station No. _________________________
Data ______________________________

Lab 9 – OO Concept (Episode III).

Task	Description	Result	Note
1	Class RectanglePlus		
2	Interface DoItPlus		
3	Robot		
4	**`moveUp()`**		
5	**`moveDown()`**		
6	**`moveLeft()`**		
7			