07_1 Interface

Object-Oriented Programming

Interfaces

- Not a class but very **similar to abstract class**
- Any class can **implement** the interface
- **Multiple inheritance** possible

```
interface Name {
    public static final Type Constant_Variable = Value;
    public abstract Type Method(Param1, Param2, ...);
}
```

- "**public static final**" for constants **can be omitted**
- "**public abstract**" for method heading **can be omitted**

Example: `Ordered` Interface (1/4)

```
public interface Ordered {

    public boolean precedes(Object other);
    public boolean follows(Object other);
    // NOTE: o1.follows(o2) == o2.precedes(o1)

}
```

Example: Ordered Interface (2/4)

```
public class Person implements Ordered {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public boolean precedes(Object other) {
        if (other instanceof Person) {
            Person otherPerson = (Person) other;
            return this.age < otherPerson.age;
        }
        return false;
    }
```

Example: `Ordered` Interface (3/4)

```
    @Override
    public boolean follows(Object other) {
        if (other instanceof Person) {
            Person otherPerson = (Person) other;
            return this.age > otherPerson.age;
        }
        return false;
    }
}
```

Example: `Ordered` Interface (4/4)

```
public class PersonOrderDemo {
    public static void main(String[] args) {
        Person person1 = new Person("Alice", 25);
        Person person2 = new Person("Bob", 30);

        System.out.println("Person1 precedes Person2: " +
                           person1.precedes(person2)); // true
        System.out.println("Person1 follows Person2: " +
                           person1.follows(person2));  // false
        System.out.println("Person2 precedes Person1: " +
                           person2.precedes(person1)); // false
        System.out.println("Person2 follows Person1: " +
                           person2.follows(person1));  // true
    }
}
```

Example: Interface Hierarchy (1/4)

```
// Base Interface
public interface Shape {
    public double calculateArea();
}

// Derived Interface
public interface ColoredShape extends Shape {
    public String getColor();
}

// More Specific Interface
public interface TexturedShape extends ColoredShape {
    public String getTexture();
}
```

Example: Interface Hierarchy (2/4)

```
public class TexturedRectangle implements TexturedShape {
    private double width;
    private double height;
    private String color;
    private String texture;

    public TexturedRectangle(double width, double height,
                             String color, String texture) {
        this.width = width;
        this.height = height;
        this.color = color;
        this.texture = texture;
    }
```

Example: Interface Hierarchy (3/4)

```
    @Override
    public double calculateArea() {
        return width * height;
    }

    @Override
    public String getColor() {
        return color;
    }

    @Override
    public String getTexture() {
        return texture;
    }
}
```

Example: Interface Hierarchy (4/4)

```
public class TexturedRectangleDemo {

    public static void main(String[] args) {
        TexturedRectangle rectangle =
                        new TexturedRectangle(5, 10, "Red", "Smooth");

        System.out.println("Area: "
                            + rectangle.calculateArea()); // Area: 50.0
        System.out.println("Color: "
                            + rectangle.getColor());     // Color: Red
        System.out.println("Texture: "
                           + rectangle.getTexture()); // Texture: Smooth
    }

}
```

The `Comparable` Interface

- In the **`java.lang`** package
- Automatically available to any program
- Only one method that must be implemented:

```
public int compareTo(Object other);
```

  - return value
    - < 0 : if the calling object "comes before" the parameter other
    - == 0 : if the calling object "equals" the parameter other
    - > 0 : if the calling object "comes after" the parameter other

Other Typical Interfaces in Java API

- `java.lang.Runnable`: having the method ‘`run()`’ to run something
- `java.util.Comparator`: having the method: ‘`int compare(a,b)`’
  - compare’s Output has the same meaning of ‘compareTo()’ in Comparable
  - i.e., returns negative int (if a < b),
  - positive int (if a > b),
  - zero (if a == b)

Example: Comparator Interface (1/3)

```
import java.util.Comparator;

class AgeComparator implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        Human h1 = (Human) o1;
        Human h2 = (Human) o2;
        return Integer.compare(h1.getAge(), h2.getAge());
    }
}
```

Example: Comparator Interface (2/3)

```
public class Human {
    private String name;
    private int age;

    public Human(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
```

Example: Comparator Interface (3/3)

```
    public static void main(String[] args) {
        Human human1 = new Human("Alice", 30);
        Human human2 = new Human("Bob", 25);
        AgeComparator comparator = new AgeComparator();
        int comparisonResult = comparator.compare(human1, human2);
        if (comparisonResult < 0) {
            System.out.println(human1.getName() + " is younger than "
                               + human2.getName());
        } else if (comparisonResult > 0) {
            System.out.println(human1.getName() + " is older than "
                               + human2.getName());
        } else {
            System.out.println(human1.getName() + " and "
                              + human2.getName() + " are the same age");
        }
    }
}
```

OUTPUT:

```
Alice is older than Bob
```

Selection Sort

position 0: [64, 25, 12, 22, 11] : minimum of {64, 25, 12, 22, 11} = 11
[11, 25, 12, 22, 64] : swap 64 and 11
position 1: [11, 25, 12, 22, 64] : minimum of {25, 12, 22, 64} = 12
[11, 12, 25, 22, 64] : swap 25 and 12
position 2 : [11, 12, 25, 22, 64] : minimum of {25, 22, 64} = 22
[11, 12, 22, 25, 64] : swap 25 and 22
position 3 : [11, 12, 22, 25, 64] : minimum of {25, 64} = 25, so no swap
result: [11, 12, 22, 25, 64]

Example: Selection Sort (1/4)

```
public class SelectionSort {
    public void sort(Comparable[] array) { // parameter: array of
                                           // interface Comparable
        int n = array.length;
        for (int i = 0; i < n - 1; i++) { // position from 0 to n-1
            int minIndex = i; // index of minimum
            for (int j = i + 1; j < n; j++) { // from i+1 to n-1
                if (array[j].compareTo(array[minIndex]) < 0) {
                    minIndex = j;
                }
            }
            swap(array, i, minIndex);
        }
    }
```

Example: Selection Sort (2/4)

```
    private void swap(Comparable[] array, int i, int j) {
        Comparable temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

Example: Selection Sort (3/4)

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

        // Integer array demo
        Comparable[] intArray = {64, 25, 12, 22, 11};
        SelectionSort sorter = new SelectionSort();
        sorter.sort(intArray);
        System.out.print("Sorted Integer Array: ");
        printArray(intArray); // 11 12 22 25 64
```

Example: Selection Sort (4/4)

```
        // String array demo
        Comparable[] stringArray =
                        {"apple", "orange", "banana", "kiwi", "grape"};
        sorter.sort(stringArray);
        System.out.print("Sorted String Array: ");
        printArray(stringArray); // apple banana grape kiwi orange

    }

    public static void printArray(Comparable[] array) {
        // print the array ...
    }
}
```

Inconsistent Interfaces

```
interface Inter1 {    int NUMBER = 25;   }
interface Inter2 {    int NUMBER = 32;   }

// Using multiple inheritance
public class InconsistentInterfaceDemo implements Inter1, Inter2 {
    public static void main(String[] argc) {
        int x = NUMBER;  // Compile error, 25? 32? which one?
    }
}

public class InconsistentInterfaceDemo implements Inter1, Inter2 {
    public static void main(String[] argc) {
        // But if we don't use NUMBER, then no compile error
    }
}
```

Polymorphism Using Interfaces (1/3)

- An instance of a class implementing an interface can be assigned to the variable of the interface

```
interface Fightable {
    public void move(int x, int y);
    public void attack(Fightable f);
}

class Fighter implements Fightable {
    public void move(int x, int y) {  //. . .   }
    public void attack(Fightable f) {  //. . .   }
}

Fighter f = new Fighter();
Fightable f = new Fighter();
```

Polymorphism Using Interfaces (2/3)

- An interface type can be used as **parameter type** in a method

```
interface Fightable {
    public void move(int x, int y);
    public void attack(Fightable f);
}

class Fighter implements Fightable {
    public void move(int x, int y) {
        // implement here
    }
    public void attack(Fightable f) {
        // implement here
    }
}
```

Polymorphism Using Interfaces (3/3)

- An interface can be used as the **return type** of a method

```
Fightable method() {
    // ...
    return new Fighter();
}
```