BIRLA INSTIITUTE OF TECHNOLOGY AND SCIENCE, PILANI
Second Semester, 2016-17
Object-Oriented Programming (CS F213)
LAB-6 [Interfaces and Nested Classes]

AGENDA

1. Interfaces in Java
2. Comparable and Comparator Interface
3. Inner classes

1 Interfaces in Java

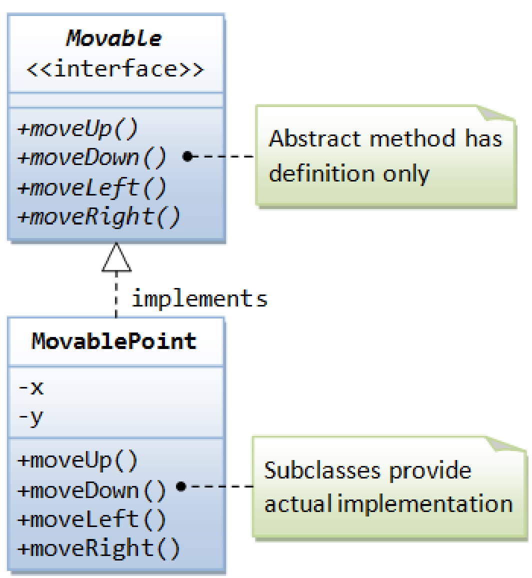
A Java interface is a *100% abstract superclass*, which define a set of methods its subclasses must support. An interface contains only public *abstract methods* (methods with signature and no implementation) and possibly *constants* (public static final variables). You have to use the keyword "interface" to define an interface (instead of keyword "class" for normal classes). The keyword public and abstract are not needed for its abstract methods, as they are mandatory.

An interface is a *contract* for what the classes can do. It, however, does not specify how the classes should do it.

Interface Naming Convention: Use an adjective (typically ends with "able") consisting of one or more words. Each word shall be initial capitalized (camel-case). For example, Serializable, Externalizable, Movable, Clonable, Runnable, etc.

Example: Movable Interface and its Implementation

Suppose that our application involves many objects that can move. We could define an interface called movable, containing the signatures of the various movement methods.



Interface Moveable.java

```
public interface Movable {
   // abstract methods to be implemented by the subclasses
   public void moveUp();
   public void moveDown();
   public void moveLeft();
   public void moveRight();
}
```

Similar to an abstract class, an interface cannot be instantiated; because it is incomplete (the abstract methods' body is missing). To use an interface, again, you must derive subclasses and provide implementation to all the abstract methods declared in the interface. The subclasses are now complete and can be instantiated.

MovablePoint.java

To derive subclasses from an interface, a new keyboard "implements" is to be used instead of "extends" for deriving subclasses from an ordinary class or an abstract class. It is important to note that the subclass implementing an interface need to override ALL the abstract methods defined in the interface; otherwise, the subclass cannot be compiled. For example,

```
public class MovablePoint implements Movable {
   // Private membet variables
   private int x, y;   // (x, y) coordinates of the point

   // Constructor
   public MovablePoint(int x, int y) {
      this.x = x;
      this.y = y;
   }


    public String toString() {
      return "Point at (" + x + "," + y + ")";
   }

   // Implement abstract methods defined in the interface Movable

   public void moveUp() {
      y--;
   }

   public void moveDown() {
      y++;
   }

   public void moveLeft() {
      x--;
   }

     public void moveRight() {
```

```
        x++;
    }
}
```

Other classes in the application can similarly implement the Movable interface and provide their own implementation to the abstract methods defined in the interface Movable.

TestMovable.java

We can also upcast subclass instances to the `Movable` interface, via polymorphism, similar to an `abstract` class.

```
public class TestMovable {
   public static void main(String[] args) {
      Movable m1 = new MovablePoint(5, 5);  // upcast
      System.out.println(m1);
      m1.moveDown();
      System.out.println(m1);
      m1.moveRight();
      System.out.println(m1);
   }
}
```

2. Implementing Multiple interfaces

As mentioned, Java supports only *single inheritance*. That is, a subclass can be derived from one and only one superclass. Java does not support *multiple inheritance* to avoid inheriting conflicting properties from multiple superclasses. Multiple inheritance, however, does have its place in programming.

A subclass, however, can implement more than one interfaces. This is permitted in Java as an interface merely defines the abstract methods without the actual implementations and less likely leads to inheriting conflicting properties from multiple interfaces. In other words, Java indirectly supports multiple inheritances via implementing multiple interfaces. For example,

```
public class Circle extends Shape implements Movable, Displayable {
// One superclass but implement multiple interfaces
   .......
}
```

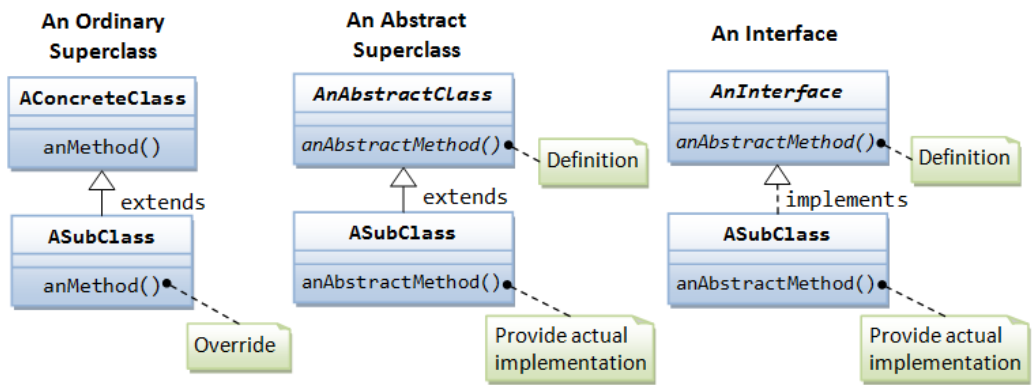
The formal syntax for declaring interface is:

```
[public|protected|package] interface interfaceName
[extends superInterfaceName] {
   // constants
   static final ...;

   // abstract methods' signature
   ...
}
```

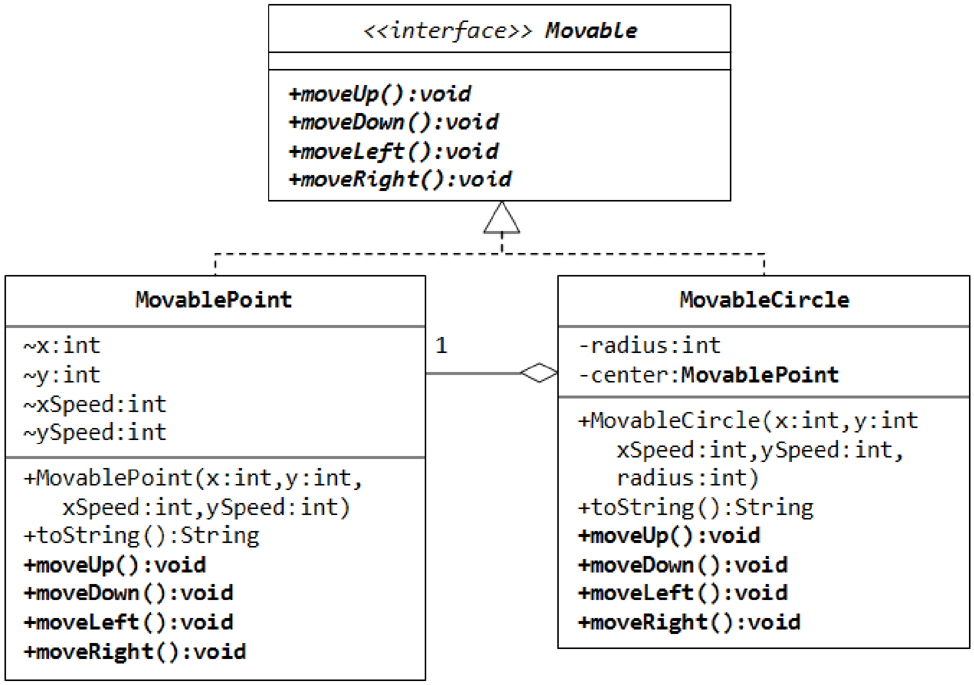
All methods in an interface shall be public and abstract (default). You cannot use other access modifier such as private, protected and default, or modifiers such as static, final.

All fields shall be public, static and final (default).
An interface may "extends" from a super-interface.
UML Notation: The UML notation uses a solid-line arrow linking the subclass to a concrete or abstract superclass, and dashed-line arrow to an interface as illustrated. Abstract class and abstract method are shown in italics.



Exercise1: Interface Movable and its implementations MovablePoint and MovableCircle

Suppose that we have a set of objects with some common behaviors: they could move up, down, left or right. The exact behaviors (such as how to move and how far to move) depend on the objects themselves. One common way to model these common behaviors is to define an *interface* called Movable, with abstract methods moveUp(), moveDown(), moveLeft() and moveRight(). The classes that implement the Movable interface will provide actual implementation to these abstract methods.
Let's write two concrete classes - MovablePoint and MovableCircle - that implement the Movable interface.



The code for the interface Movable is straight forward.

```
public interface Movable {  // saved as "Movable.java"
   public void moveUp();
   //Complete the implementation
}
```

For the MovablePoint class, declare the instance variable x, y, xSpeed and ySpeed with package access as shown with '~' in the class diagram (i.e., classes in the same package can access these variables directly). For the MovableCircle class, use a MovablePoint to represent its center (which contains four variable x, y, xSpeed and ySpeed). In other words, the MovableCircle composes a MovablePoint, and its radius.

```
//Complete the implementation
public class MovablePoint implements Movable {

   // instance variables
   int x, y, xSpeed, ySpeed;     // package access

   // Constructor
   public MovablePoint(){}
   public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
      this.x = x;
   }

   // Implement abstract methods declared in the interface Movable

   public void moveUp()  {
      y -= ySpeed;   // y-axis pointing down for 2D graphics
   }
}

//Complete the implementation
```

```
public class MovableCircle implements Movable {
   // instance variables
   private MovablePoint center;   // can use center.x, center.y directly
                                  //  because they are package accessible
   private int radius;

   // Constructor
   public MovableCircle(int x, int y, int xSpeed, int ySpeed, int radius) {
      // Call the MovablePoint's constructor to allocate the center instance.
     super();
     center = new MovablePoint(x, y, xSpeed, ySpeed);
     this.radius = radius;
   }

   // Implement abstract methods declared in the interface Movable
   @Override
   public void moveUp() {
      center.y -= center.ySpeed;
   }
}
```

Q.1 Write a test program and try out these statements:

```
Movable m1 = new MovablePoint(5, 6, 10);     // upcast
System.out.println(m1);
m1.moveLeft();
System.out.println(m1);

Movable m2 = new MovableCircle(2, 1, 2, 20); // upcast
System.out.println(m2);
m2.moveRight();
System.out.println(m2);
```

Q.2 Modify your code of the MovablePoint class, so that two MovablePoint objects can be compared on the basis of their x and y coordinates.

Q.3 Modify your code of the MovableCircle class so that two MovableCircle objects can be compared on the basis of their (a) radius (b) center and (c) radius and center.

3. Nested classes

1. Classes that are declared inside the body of a class are called "nested classes".
2. The following are the main reasons behind the concept of nested classes in Java:
    a. Grouping of logical classes
        - When designing a class, we always try to create well-focused classes - with a unique behavior and purpose. Let us imagine we have designed classes A and B on the same principle. Later we found, class B can only be used by class A. In such cases, we can simply put class B inside class A.

    b. Encapsulation
        - By writing nested classes, we have hidden them from the world and made them available only to their enclosing class.

c. Maintainability and re-usability of code
    - Encapsulation brings the beauty of maintainability of code. In our earlier example, we can write another class B which is visible t o the entire world. This has nothing to do with the one already present inside class A.

3. Nested class is of 2 kinds:
    a. Inner class
    b. Static nested class
4. Inner class is of 3 types:
    a. Inner class (Member class)
    b. Method-local inner class
    c. Anonymous inner class
5. Nested class behaves just like any other member of its enclosing(outer) class.
6. It has access to all the members of its enclosing class.

4. Inner Class

1. We define the term "inner class" to the nested class that is:
    a. declared inside the body of another class.
    b. not declared inside a method of another class.
    c. not a static nested class.
    d. not an anonymous inner class.
2. An example:

```
class Outer{
    class Inner{
    }
}
```

3. When we compile the above code we get 2 class files:
    - Outer.class
    - Outer$Inner.class
4. Notice that inner class is tied to its outer class though it is still a separate class.
5. An inner class cannot have any kind of static code including the public static void main(String[] args).
6. Only classes with "public static void main(String[] args)" can be called using "java" command.
7. In our earlier example, Inner class didn't have a static main method. So, we can't call java Outer$Inner!
8. The inner class is just like any other member of its enclosing class.
9. It has access to all of its enclosing class' members including private.

5. Instantiating an Inner Class

1. Since inner class can't stand on its own, we need an instance of its outer class to tie it.
2. There are 2 ways to instantiate an instance of inner class:
    a. From within its outer class
    b. From outside its outer class

```
class Outer{

   Inner i1 = new Inner();
```

```
    private String s = "Outer string"; //Outer instance variable

    void getS(){ System.out.println(s); }

    void getInnerS(){ System.out.println(i1.s); }

    class Inner{
        //Inner instance variable, initialized
        private String s = "Inner string";
        void getS(){
            System.out.println(s);
        }
        void getOuterS(){
            System.out.println(Outer.this.s);
        }
    }

    public static void main(String[] args){
        Outer o = new Outer();
        //can also be new Outer().new Inner();
        Outer.Inner oi = o.new Inner();
        o.getS();
        oi.getS();
        o.getInnerS();
        oi.getOuterS();
    }
}
```

6. Method-Local Inner Classes

1. A method-local inner class is defined within a method of the enclosing class.
2. For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but after the class definition code.
3. A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked final.
4. The only modifiers you can apply to a method-local inner class are abstract and final. (Never both at the same time, though.)

Excercise:2 [Using Interface and ArrayList]

a. Define an interface named 'BinaryInterface' with the following abstract method

```
+ toBinary (N: int ): void
```

/* It convert Decimal Number into binary. It stores binary representation of decimal number into ArrayList. In ArrayList each index stores either 0 or 1. Suppose if N is +8. It stores [0, 0, 0, 0, 1, 0, 0, 0] in an ArrayList and N range is from -127 to +127.

Note: The number N is represented in 8-bit format. */

b. Define the class name 'OneComplement' which encapsulates One's complement binary representation of decimal number. This class also implements interface 'BinaryInterface'.The class 'OneComplement' is described as follows:

```
Attribute:
+ Arr : ArrayList<Integer> // It stores the one’s complement binary representation of decimal number.
Operation:
+ toBinary (N: int ): void
/* This method converts the decimal number ‘N’ into one’s compliment binary
representation in an ArrayList Arr. */
```

c. Define the class name ‘SignedMagnitude’ which encapsulates One’s complement binary representation of decimal number. This class also implements interface ‘BinaryInterface’. The class ‘SignedMagnitude’ is described as follows:

```
Attribute:
+ Arr : ArrayList<Integer> // It stores the signed magnitude binary representation of decimal number.
Operation:
+ toBinary (N: int ): void
/* This method converts the decimal number ‘N’ into signed magnitude binary
representation in an ArrayList Arr. */
```

d. Define the class name ‘TwoComplement’ which encapsulates One’s complement binary representation of decimal number. This class also implements interface ‘BinaryInterface’.The class ‘TwoComplement’ is described as follows:

```
Attribute:
+ Arr : ArrayList<Integer> // It stores the two’s complement binary representation of decimal number.
Operation:
+ toBinary (N: int ): void
/* This method converts the decimal number ‘N’ into two’s compliment binary
representation in an ArrayList Arr. */
```

e. Write the suitable Driver class named BinaryTest to show the Runtime Polymorphism Approach.

Sample Output:
One'sComplement of n = -16 is [1, 1, 1, 0, 1, 1, 1, 1]
Two'sComplement of n = -16 is [1, 1, 1, 1, 0, 0, 0, 0]
Sign Magnitude of n = -16 is [1, 0, 0, 1, 0, 0, 0, 0]