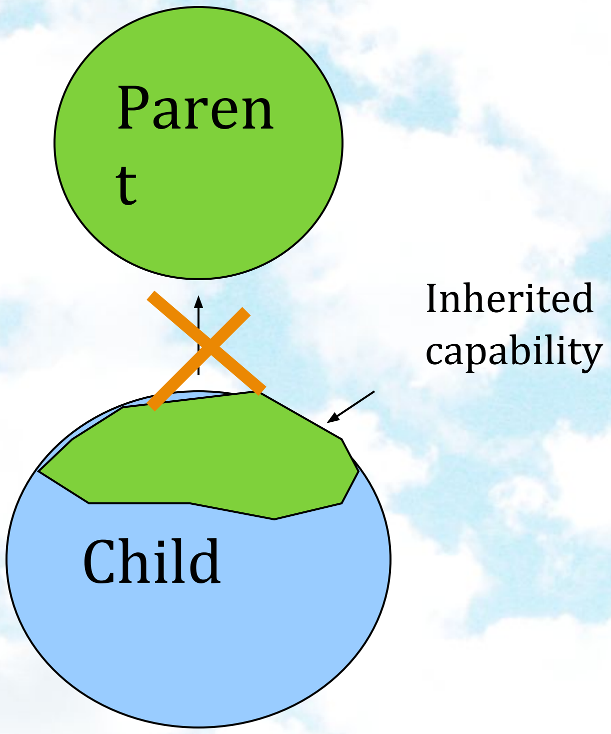
CSE2101: Object Oriented Programming-II (Java)

Lecture 11

Final and Abstract Classes

Restricting Inheritance



Final Members: A way for Preventing Overriding of Members in Subclasses

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword "final" as a modifier. For example:
  - final int marks = 100;
  - final void display();
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

Final Classes: A way for Preventing Classes being extended

- We can prevent an inheritance of classes by other classes by declaring them as final classes.
- This is achieved in Java by using the keyword final as follows:

```
final class Marks
{ // members
}
final class Student extends Person
{ // members
}
```

- Any attempt to inherit these classes will cause an error.

Abstract Classes

- When we define a class to be “final”, it cannot be extended. In certain situation, we want the properties of classes to be always extended and used. Such classes are called Abstract Classes.
- An *Abstract* class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package.
- Java Abstract class is used **to provide common method implementation to all the subclasses or to provide default implementation**.

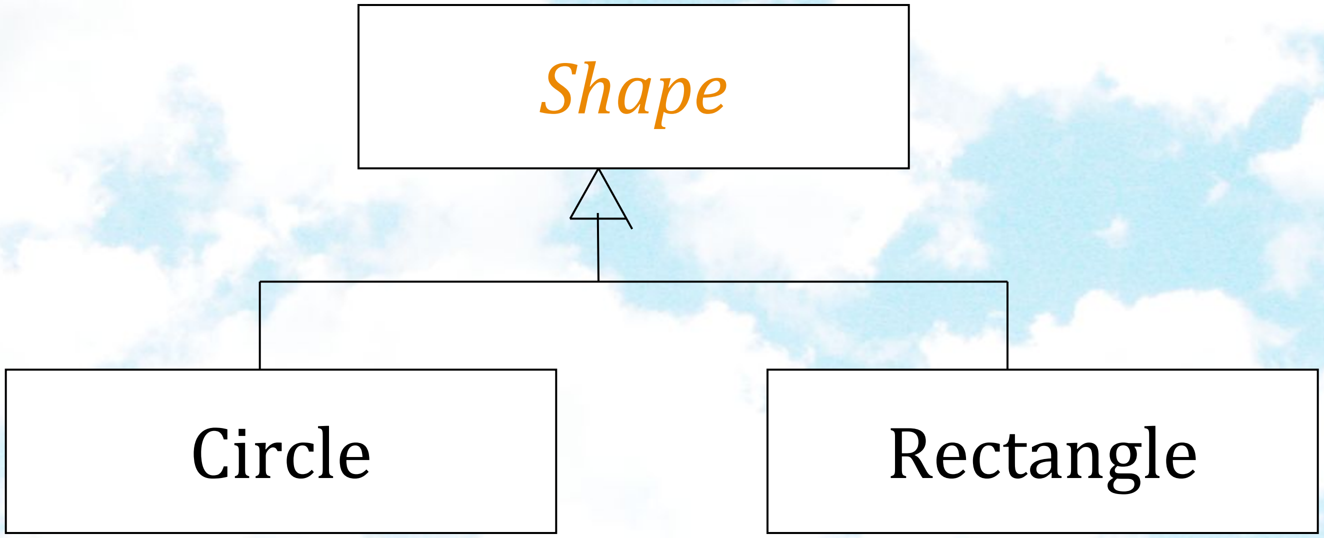
Abstract Class Syntax

```
abstract class ClassName
{
         . . . ...
         abstract Type MethodName1();
         ......
         Type Method2()
         {
           // method body
         }
}
```

- When a class contains one or more abstract methods, it should be declared as abstract class.
- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructors or abstract static methods.

Abstract Class -Example

- Shape is a abstract class.



The Shape Abstract Class

```
public abstract class Shape {
        public abstract double area();
        public void move() { // non-abstract
method
         // implementation
     }
}
```

- Is the following statement valid?
  - Shape s = new Shape();
- No. It is illegal because the Shape class is an abstract class, which cannot be instantiated to create its objects.

Abstract Classes

```
public Circle extends Shape {
        protected double r;
        protected static final double PI =3.1415926535;
        public Circle() { r = 1.0; )
        public double area() { return PI * r * r; }
…
}
public Rectangle extends Shape {
        protected double w, h;
        public Rectangle() { w = 0.0; h=0.0; }
        public double area() { return w * h; }
}
```

Abstract Classes Properties

- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementation them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.

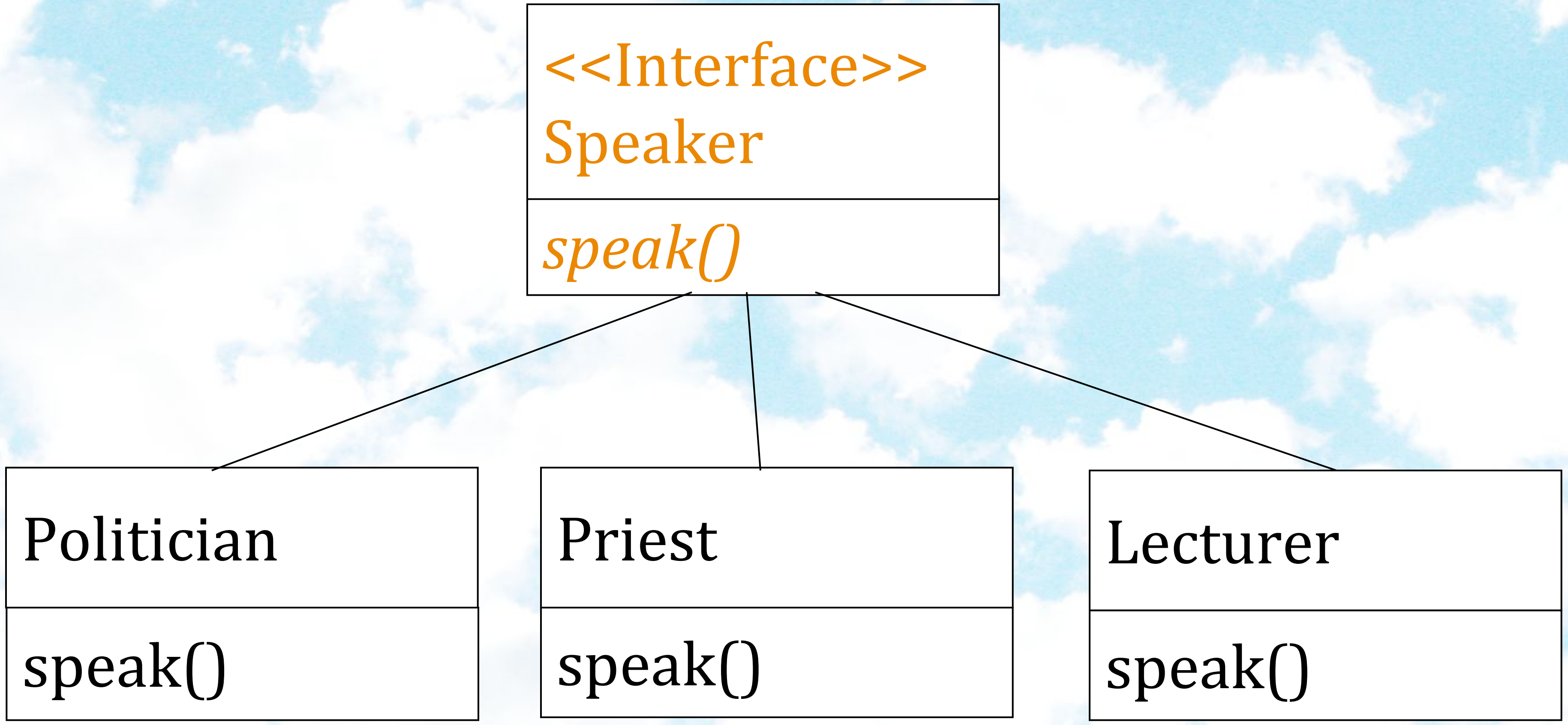
Interfaces

Design Abstraction and a way for loosing realizing Multiple Inheritance

Interfaces: An informal way of realizing multiple inheritance

- *Interface* is a conceptual entity similar to a Abstract class.
- Can contain only constants (final variables) and abstract method (no implementation) - Different from Abstract classes.
- It is the responsibility of the class that implements an interface to supply the code for methods.
- Use when a number of classes share a common interface.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realizing multiple inheritance in Java.

Interface - Example



Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {
      // Constant/Final Variable Declaration
      // Methods Declaration – only method
body
}
```

- Example:

```
interface Speaker {
      public void speak( );
}
```

Implementing Interfaces

- Interfaces are used like super-classes who properties are inherited by classes. This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, …]
{
        // Body of Class
}
```

Implementing Interfaces Example

```
class  Politician implements Speaker {
            public void speak(){
                  System.out.println("Talk politics");
            }
}
```

```
class  Priest implements Speaker {
            public void speak(){
                  System.out.println("Religious Talks");
            }
}
```

```
class  Lecturer implements Speaker {
            public void speak(){
                  System.out.println("Talks Object Oriented Design and 
Programming!");
            }
}
```

Extending Interfaces

- Like classes, interfaces can also be extended. The new sub-interface will inherit all the members of the superinterface in the manner similar to classes. This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends
InterfaceName1 {
      // Body of InterfaceName2
}
```

Inheritance and Interface Implementation

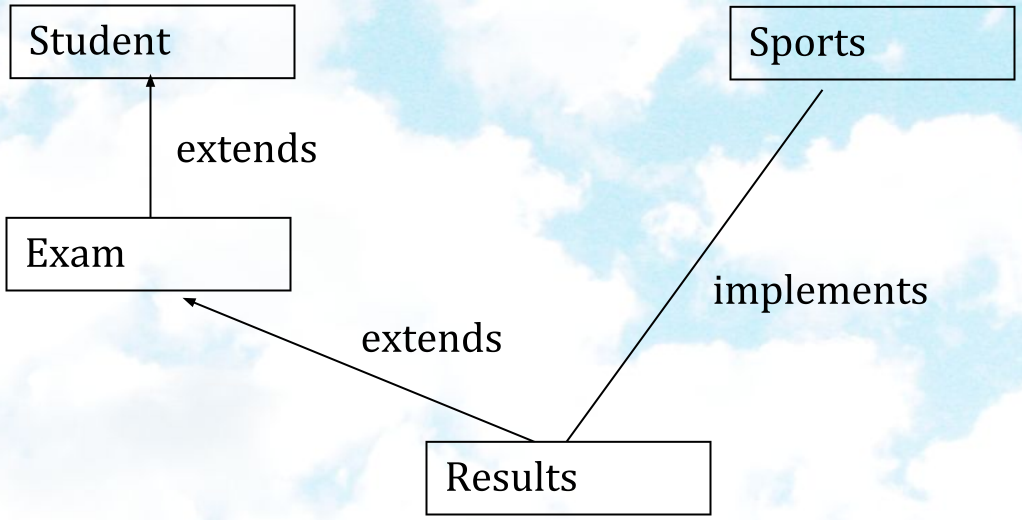
- A general form of interface implementation:

```
class ClassName extends SuperClass implements
InterfaceName [, InterfaceName2, …]
{
        // Body of Class
}
```

- This shows a class can be extended to another class while implementing one or more interfaces. It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

Student Assessment Example

- Consider a university where students who participate in the national games or Olympics are given some grace marks. Therefore, the final marks awarded = Exam_Marks + Sports_Grace_Marks. A class diagram representing this scenario is as follow:



Software Implementation

```
class Student
{
      // student no and access methods
}
interface Sport
{
      // sports grace marks (say 5 marks) and abstract methods
}
class Exam extends Student
{
      // example marks (test1 and test 2 marks) and access methods
}
class Results extends Exam implements Sport
{
      // implementation of abstract methods of Sport interface
      // other methods to compute total marks = test1+test2+sports_grace_marks;
      // other display or final results access methods
}
```

Extending interfaces – about constants (1)

- An extended interface inherits all the constants from its superinterfaces
- Take care when the subinterface inherits more than one constants with the same name, or the subinterface and superinterface contain constants with the same name — always use sufficient enough information to refer to the target constants

Tedious Details (1)

- When a class inherits two or more constants with the same name

E.g.

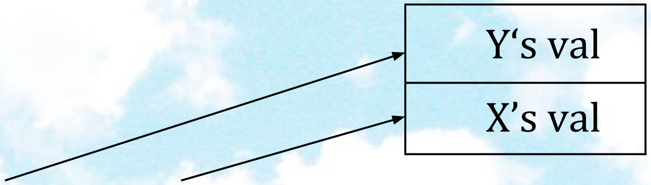
```
interface A {
    int val = 1;
}
interface B {
    int val = 2;
}
class C implements A, B {

        ...
        ...
        System.out.println("A.val = "+ A.val);
System.out.println("B.val = "+ B.val);
}
```

Tedious Details (2)

- If a superinterface and a subinterface contain two constants with the same name, then the one belonging to the superinterface is **hidden**

1. in the subinterface
   - access the subinterface-version constants by directly using its name
   - access the superinterface-version constants by using the superinterface name followed by a dot and then the constant name

     E.g
     ```
     interface X {
        int val = 1; }
     interface Y extends X{
        int val = 2;
        int sum = val + X.val; }
     ```



2. outside the subinterface and the superinterface
   - you can access both of the constants by explicitly giving the interface name.

     E.g. in previous example, use Y.val and Y.sum to access constants val and sum of interface Y, and use X.val to access constant val of interface X.

Tedious Details (3)

- When a superinterface and a subinterface contain two constants with the same name, and a class implements the subinterface
  - the class inherits the subinterface-version constants as its static fields. Their access follow the rule of class's static fields access.

E.g

```
class Z implements Y { }
  //inside the class
  System.out.println("Z.val:"+val);   //Z.val = 2
  //outside the class
  System.out.println("Z.val:"+Z.val); //Z.val = 2
```

  - object reference can be used to access the constants
    - subinterface-version constants are accessed by using the object reference followed by a dot followed by the constant name
    - superinterface-version constants are accessed by explicit casting

E.g.

```
Z v = new Z( );
System.out.print( "v.val = " + v.val
                 +", ((Y)v).val = " + ((Y)v).val
                      +", ((X)v).val = " + ((X)v).val );
```

output: v.val = 2, ((Y)v).val = 2, ((X)v).val = 1

Interfaces and abstract classes

- Why bother introducing two concepts: abstract class and interface?

```
abstract class Comparable  {
   public abstract int compareTo (Object otherObject);
}
class Employee extends Comparable  {
    pulibc int compareTo(Object otherObject) { . . . }
}
 ---------------------------------------------------------
public interface Comparable {
    int compareTo (Object otherObject)
}
class Employee implements Comparable  {
    public int compareTo (Object otherObject) { . . . }
}
```

- A class can only extend a single abstract class, but it can implement as many interfaces as it wants
- An abstract class can have a partial implementation, protected parts, static methods and so on, while interfaces are limited to public constants and public methods with no implementation

Interfaces and Software Engineering

- *Interfaces*, like abstract classes and methods, provide templates of behaviour that other classes are expected to implement.
- Separates out a design hierarchy from implementation hierarchy. This allows software designers to enforce/pass common/standard syntax for programmers implementing different classes.
- Pass method descriptions, not implementation
- Java allows for inheritance from only a single superclass. *Interfaces* allow for *class mixing*.
- Classes *implement* interfaces.

Thank you