Object-Oriented Programming and Design

Part III

Topics in this section

- Abstract classes review
- Interfaces, multiple inheritance
- Inner classes introduced

Abstract classes and methods

- An abstract class may contain abstract methods
- An abstract method is a method with no body (i.e., simply a semicolon after the parameter list)
- An abstract method constitutes a protocol or contract, that is, regular or non-abstract subclasses are required to implement the abstract methods of superclasses
- Thus, if a superclass has an abstract method, it guarantees that all subclasses (even future subclasses) implement this method
- For example, an abstract toString() method in a class forces all its subclasses to implement a toString() method

Interactive programming exercises: Wind2.java revisited

Interfaces (1)

- Defined using the keyword “interface” instead of “class” -- Both interface and class names are types in Java
- Interfaces contain
  - ✍ **`Only abstract`** methods
  - ✍ **`public static final`** fields (i.e., constants)
- All members of an **`interface`** are **`public by default`**
- A Java interface can be used in the same way as a Java class
- Just like abstract classes, interfaces cannot be instantiated

Interfaces (2)

- Interfaces are basically abstract classes except
  - ✍ All methods in an **`interface`** are **`abstract`**
  - ✍ An **`abstract`** class may contain non-abstract methods
  - ✍ Thus, some methods of an **`abstract class`** may be implemented
  - ✍ No methods of an **`interface`** may be implemented
- A class **`implements`** an **`interface`** by
  - ✍ Declaring that it **`implements`** the **`interface`**

    `class X implements I { … }`
  - ✍ Defining (or providing) implementations of all the **`interface`** methods

Interfaces (3)

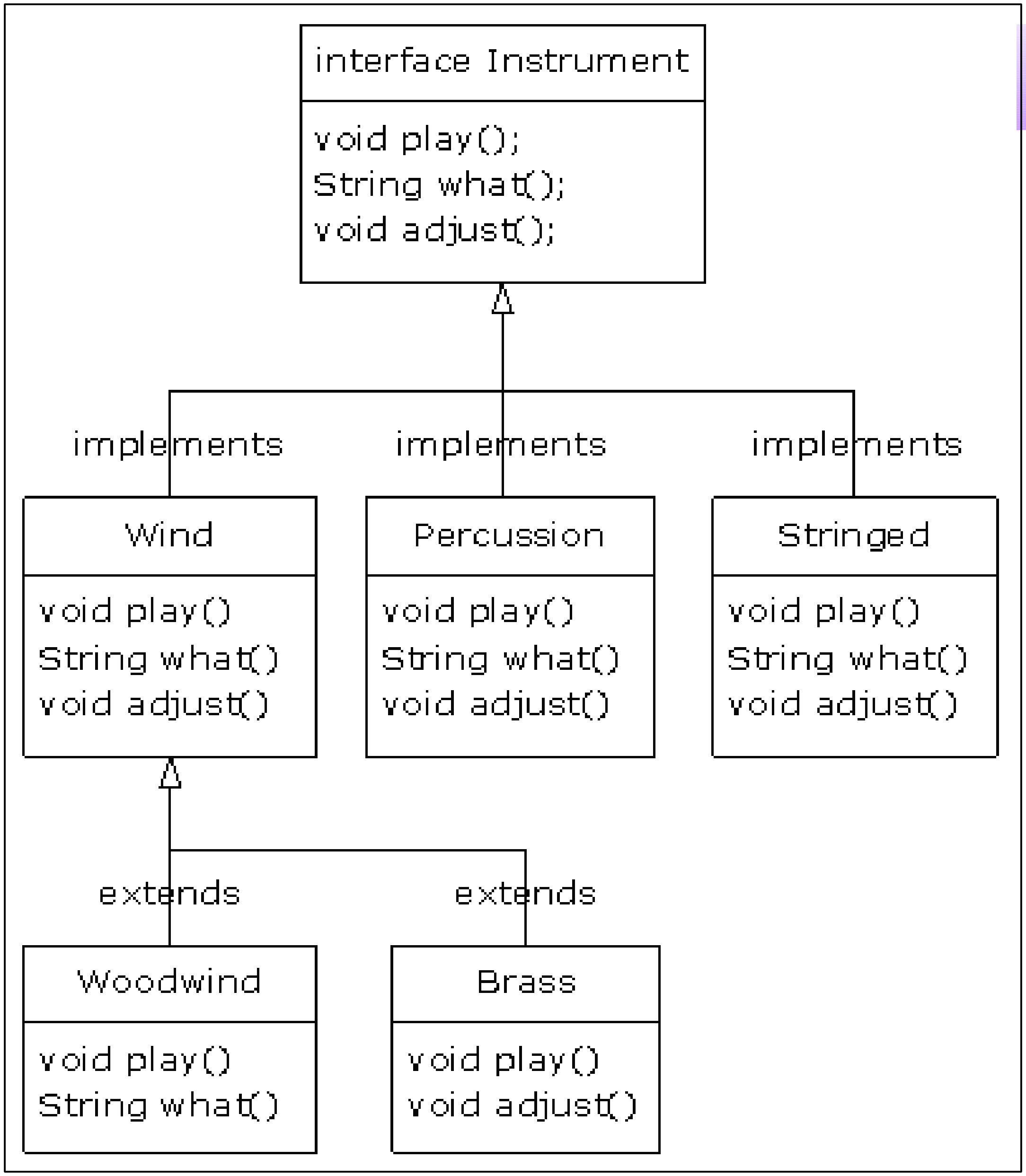
- An *interface* may **extend** one or more interfaces

```
interface Stack extends List, Comparable { … }
interface Container extends Collection { … }
```

- When a class **implements** an interface method, it must implement its exact signature
- Interfaces form their own inheritance hierarchy
- A *class* may implement multiple interfaces and extend one or zero classes
  - This is essentially multiple inheritance

```
class X implements I1, I2 { … }
class X extends A implements I1, I2 { … }
```

- The relationships induced by **extends** and **implements** are all is-a relationships (so we can do upcasting!)



Interactive programming exercise: Music.java

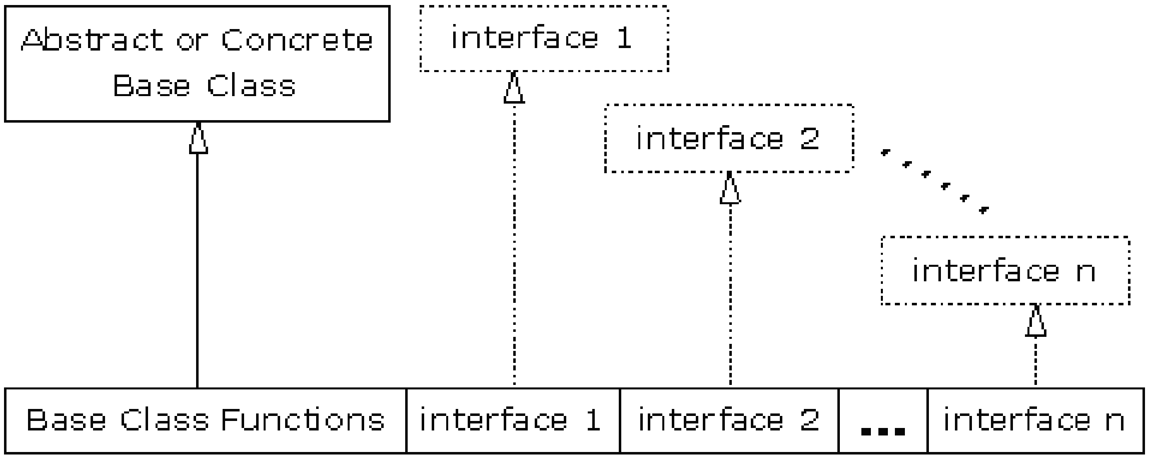
Defining and implementing interface Comparable

```
public interface Comparable {
  int compareTo(Object x);  // public by default!
}

public abstract class Shape implements Comparable {
  public abstract double area();
  public int compareTo(Object x) { // provides implementation
    Shape s = (Shape)x;
    double diff = area() – s.area();
    if (diff == 0) return 0;
    else if (diff < 0) return -1;
    else return 1;
  }
}
```

Multiple inheritance in more detail....

- We can have multiple inheritance in Java without some of the sticky issues faced in other languages (such as C++)
- There is only one implementation, so we know which method should be run

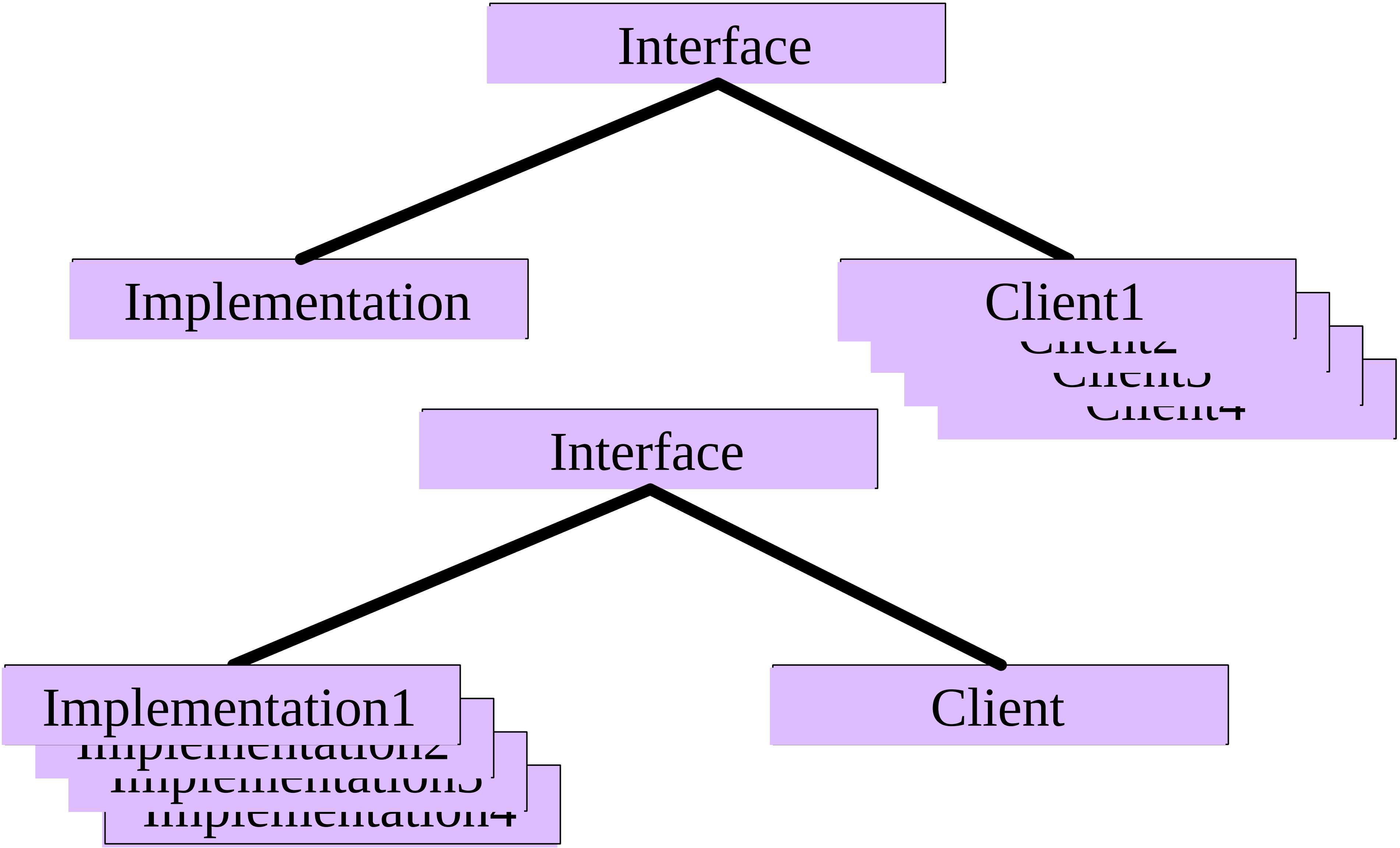


Interactive programming exercise: Adventure.java

Separating the What from the How

- *Complexity* is a big problem in software engineering
- We can control complexity by:
  - Separating concerns
  - Breaking software into smaller, simpler pieces
  - Making sure that each piece knows only *what* other pieces do, not *how* they do it
    - Abstraction, information hiding, encapsulation
    - High coupling within components
    - Low coupling among components
- Benefits of this approach:
  - ease of use
  - ease of modification
  - Ease of maintenance and evolution
- Java interfaces separate the *what* from the *how* (*reduce coupling*)
- Java classes *encapsulate* all that is necessary to implement an interface (*increase cohesion*)

Interface relationships



Interface

Implementation

Client1

Client2

Client3

Client4

Interface

Implementation1

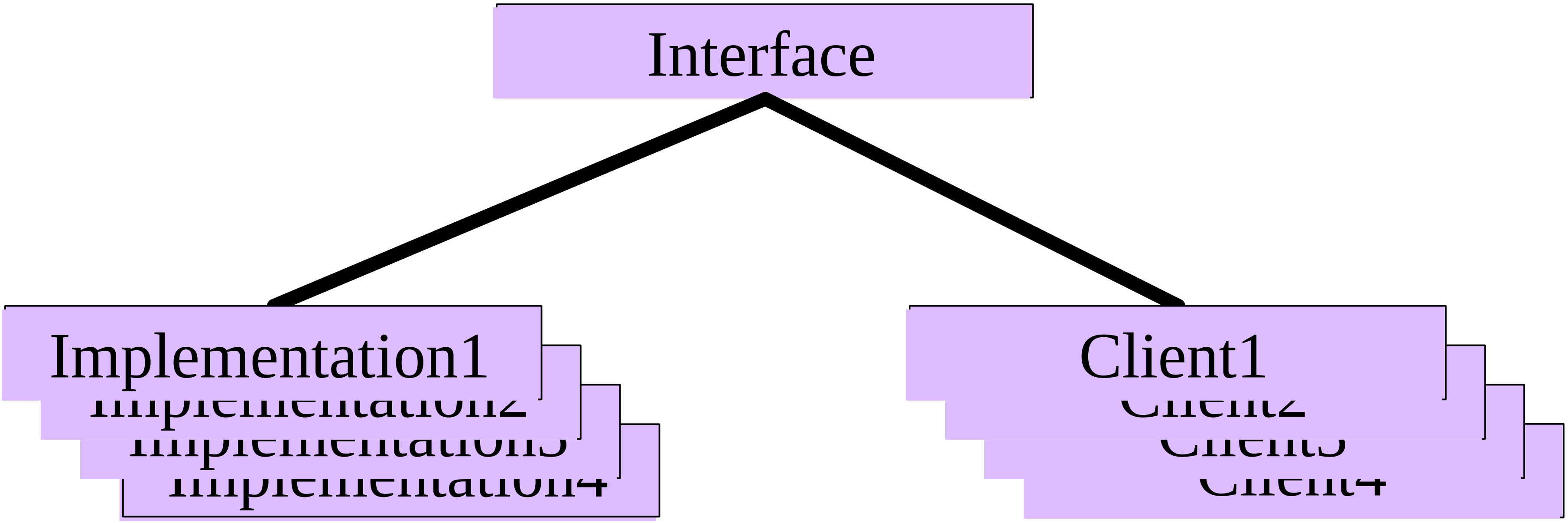
Implementation2

Implementation3

Implementation4

Client

Interface relationships



Interface

Implementation1

Implementation2

Implementation3

Implementation4

Client1

Client2

Client3

Client4

- Sorting interface
  - Implemented by different sorting algorithms (e.g., Quicksort, Heapsort, Insertionsort, Bubblesort, Mergesort, Radixsort)
  - Used by different clients to sort Strings, integers, doubles, dates, records

General hint for design

- Should you use an abstract or an interface?
- An interface gives you the benefits of a class and an interface, so use an interface if you can! (but use an abstract class if you need some implementations or non-static final fields)

Inner Classes -- introduced

- You can place class definitions inside other class definitions – called an **Inner class**
- More than just a simple code-hiding mechanism!
- If just hiding was an issue, we would just make a class be friendly so that only classes in the package would see it
- It knows about and can communicate with surrounding classes
- Inner classes are important when you want to upcast to a base class or interface
- We will see inner classes soon when you learn about iterators

Interactive programming exercise: Parcel1.java