UBC OKANAGAN

COSC 121
Computer Programming II

Abstract Classes and Interfaces

Part 1/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Previous Pre-recorded Lecture

Students' led Q/As about the previous lecture:

Polymorphism

- Generic programming
- `instanceof` operator
- Object casting
- `Object`'s `equals` method

Outline

Today:

- Abstract classes
- User-defined interfaces

Next lecture:

- More on user-defined interfaces
  - default, static, and private methods
- Java standard interfaces
  - `Comparable` interface
  - `Cloneable` interface
    - Shallow vs. deep cloning.
- Interfaces or abstract classes?

Abstract Classes

Abstract Classes and Methods

Abstract classes are usually used to define **common features** to their **subclasses**.

- You **cannot** create objects of an abstract-class type.
- An abstract class may contain ***abstract methods.***
- Syntax: `abstract class myClass{…}`

Abstract method have a header but **not a body**.

- Syntax: `abstract void myMethod();`

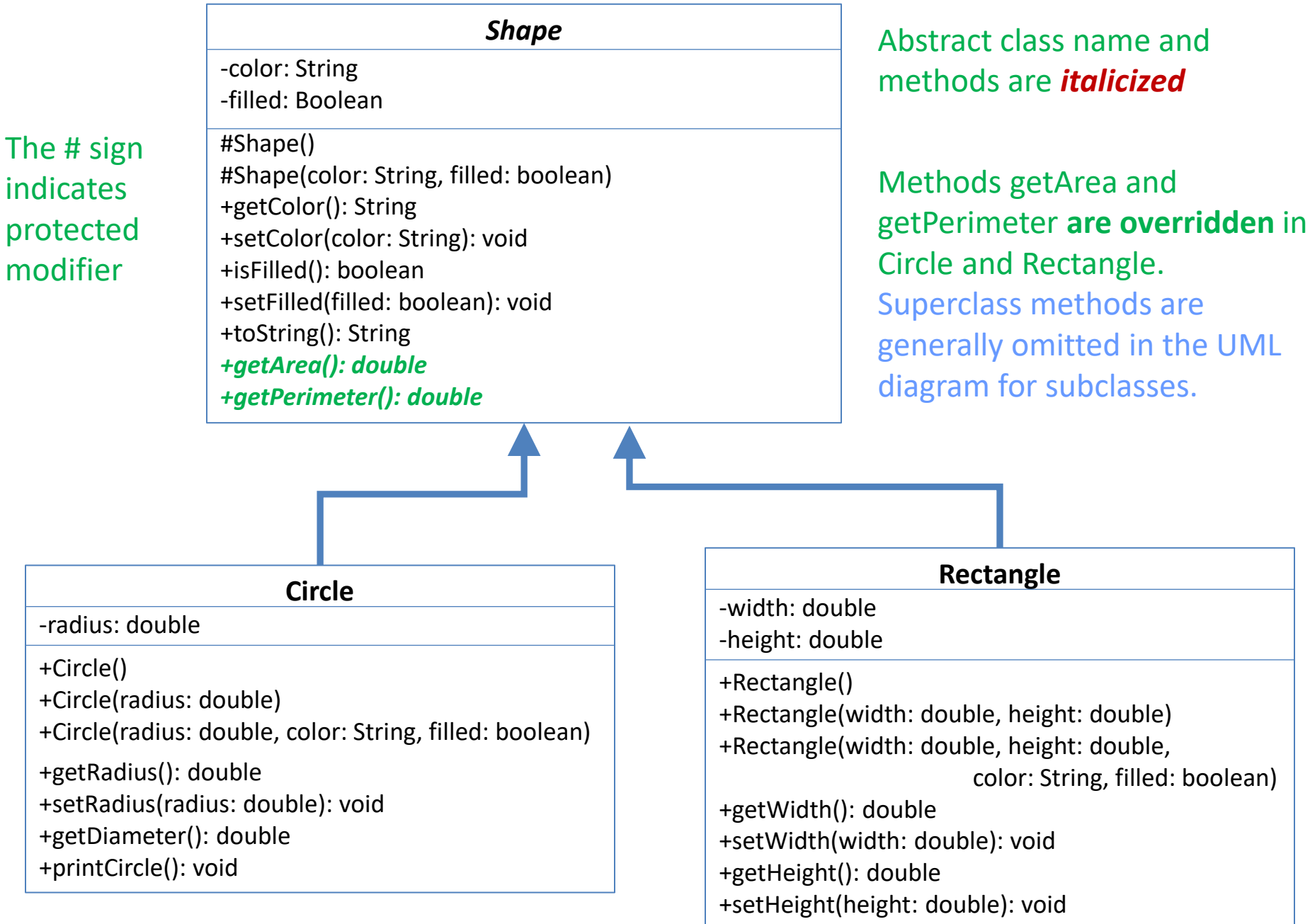
Concrete classes (i.e. not abstract) cannot contain abstract methods.

- In a concrete subclass extended from an abstract class, all the abstract methods **must be implemented (overridden)**, even if they are not used in the subclass.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract.

Why abstract methods?

- Abstract methods are used to specify **common behavior** for subclasses that have **different implementation** of those methods.

Abstract Class Example



Abstract Class Example, cont.

In this example, we have two abstract methods `getArea()` and `getPerimeter()`. Both must be implemented in a concrete subclass.

```
public abstract class Shape {
    //attributes
    private String color;
    private boolean filled;
    //constructors
    protected Shape(){this("White", true);}
    protected Shape(String color, boolean filled){
        setColor(color);
        setFilled(filled);
    }
    //methods
    public String getColor(){return color;}
    public void setColor(String color){this.color = color;}
    public boolean isFilled(){return filled;}
    public void setFilled(boolean f){filled = f;}
    public String toString(){return "Color: " + color + ". Is filled? " + filled;}
    public abstract double getArea();
    public abstract double getPerimeter();
}
```

Abstract Class Example, cont.

Here, we see that we had to implement `getArea()` and `getPerimeter()` in a concrete subclass, i.e. Circle in this case.

```
public class Circle extends Shape{
    //attributes
    private double radius;
    //constructors
    public Circle(){this(10, "White", true);}
    public Circle(double r){this(r, "While", true);}
    public Circle(double radius, String color, boolean filled){
        super(color, filled);
        setRadius(radius);
    }
    //methods
    public double getRadius(){return radius;}
    public void setRadius(double radius){this.radius = radius;}
    public double getDiameter(){return 2 * getRadius();}
    public void printCircle(){System.out.println(super.toString() + getColor());}
    public double getArea(){return Math.PI * getRadius() * getRadius();}
    public double getPerimeter(){return 2 * Math.PI * getRadius();}
}
```

Abstract Class Example, cont.

Here, we see that we had to implement `getArea()` and `getPerimeter()` in a concrete subclass, i.e. Rectangle in this case.

```
public class Rectangle extends Shape{
    //attributes
    private double width, height;
    //constructors
    public Rectangle() {this(1, 1, "White", true);}
    public Rectangle(double width, double height) {
        this(width, height, "White", true);}
    public Rectangle(double width, double height, String color, boolean filled) {
        super(color, filled);
        this.setWidth(width);
        this.setHeight(height);
    }
    //methods
    public double getHeight() {return height;}
    public void setHeight(double height) {this.height = height;}
    public double getWidth() {return width;}
    public void setWidth(double width) {this.width = width;}
    public double getArea() {return getHeight() * getWidth();}
    public double getPerimeter() {return 2 * (getHeight() + getWidth());}
}
```

Notes about abstract classes/methods

A class that contains abstract methods must be abstract.

- it is also possible to define abstract class with no abstract methods.

A subclass can be abstract even if its superclass is concrete.

- Example: `Object` class is concrete, but its subclasses may be abstract.

An abstract subclass can override a concrete method from its superclass to define it abstract.

- This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass.

You can define constructors in an abstract class, which are invoked in the constructors of its subclasses.

- Remember that you cannot create objects of abstract classes.

You ca**n create reference variables** of an abstract-class type

- Example: `Shape shp = new Circle();`

Interfaces

Interfaces

An **interface** is a **class-like construct** that contains

- **constants**
- **methods: *abstract***, ***default***, or ***static***

An interface defines common behavior for classes (**including unrelated classes**) and provide default implementation of some of that behavior.

- For example, you can specify that the unrelated objects are ***comparable***, ***edible***, and/or ***cloneable*** using interfaces.

Abstract classes vs Interfaces

In many ways, an interface is **similar to an abstract class.** But remember that an **interface is not a class**.

- For example, interfaces cannot have constructors or instance variables.

Furthermore, conceptually they are "*not parents*" to implementing classes but rather define **common behavior** and allow **multiple inheritance**.

- *As we will see shortly*

Similar to abstract classes:

- You **cannot create objects** of an interface type.
- You **CAN create reference variables** of an interface type.

We are going to revisit this comparison again later.

How to Define an Interface

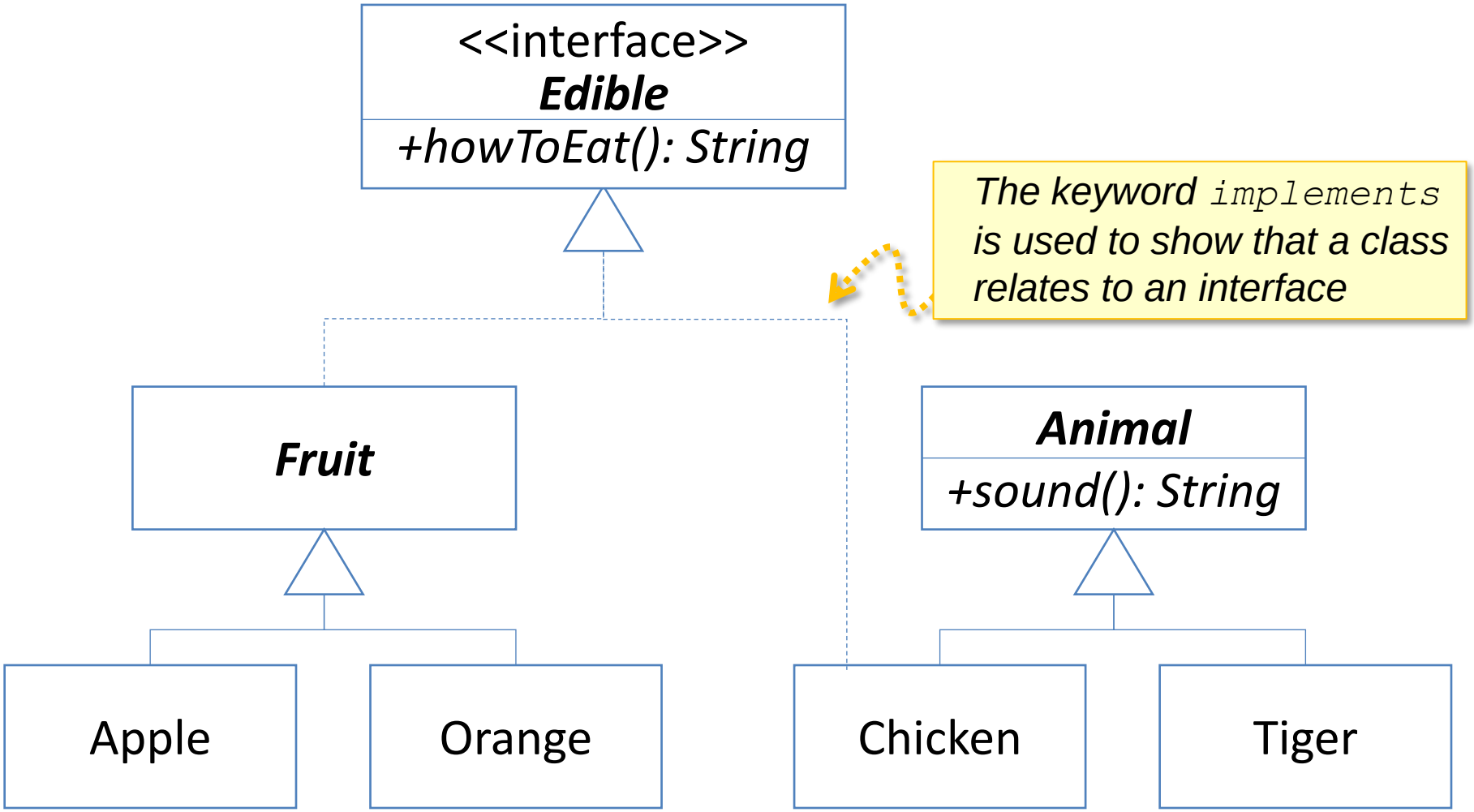
Interfaces have **zero or more** of the following items:

```
public interface somename {
   constants
   abstract methods
   default methods          //introduced in Java 8
   static methods           //introduced in Java 8
   private methods          //introduced in Java 9
   private static methods //introduced in Java 9
}
```

Example: Edible interface to specify whether an object is edible.

```
public interface Edible {
   public abstract String howToEat();
}
```

Example



Notation: interface name and abstract methods and classes are italicized. The dashed lines and hollow triangles are used to point to interfaces.

```java
public class TestEdible {
    public static void main(String[] args) {
        Object[] objects = { new Tiger(), new Chicken(), new Apple() };
        for (int i = 0; i < objects.length; i++) {
            if (objects[i] instanceof Edible)
                System.out.println(((Edible) objects[i]).howToEat());
            if (objects[i] instanceof Animal)
                System.out.println(((Animal) objects[i]).sound());
        }
    }
} //End of TestEdible
abstract class Animal {
    public abstract String sound();
}
class Chicken extends Animal implements Edible {
    public String howToEat() {return "Chicken: Fry it";}
    public String sound() {return "Chicken: cock-a-doodle-doo";}
}
class Tiger extends Animal {
    public String sound() {return "Tiger: RROOAARR";}
}
abstract class Fruit implements Edible {
    // class members & constructors omitted here
}
class Apple extends Fruit {
    public String howToEat() {return "Apple: Make apple cider";}
}
class Orange extends Fruit {
    public String howToEat() {return "Orange: Make orange juice";}
}
```

Notes on code in previous slide

1) The statement:

```
Object[] objects={new Tiger(),new Chicken(),new
Apple()};
```

is equivalent to

```
Object[] objects = new Object[3];
objects[0] = new Tiger();
objects[1] = new Chicken();
objects[2] = new Apple();
```

2) The above array uses references of the **`Object`** type, which means it can contains instances of any class type

3) Since **`howToEat()`** and **`sound()`** are not methods of **`Object`**, elements retrieved from the array had to be cast to the appropriate types using **`Edible`** or **`Animal`**.

- e.g., `((Edible) Objects[i]).howToEat();`

Some rules

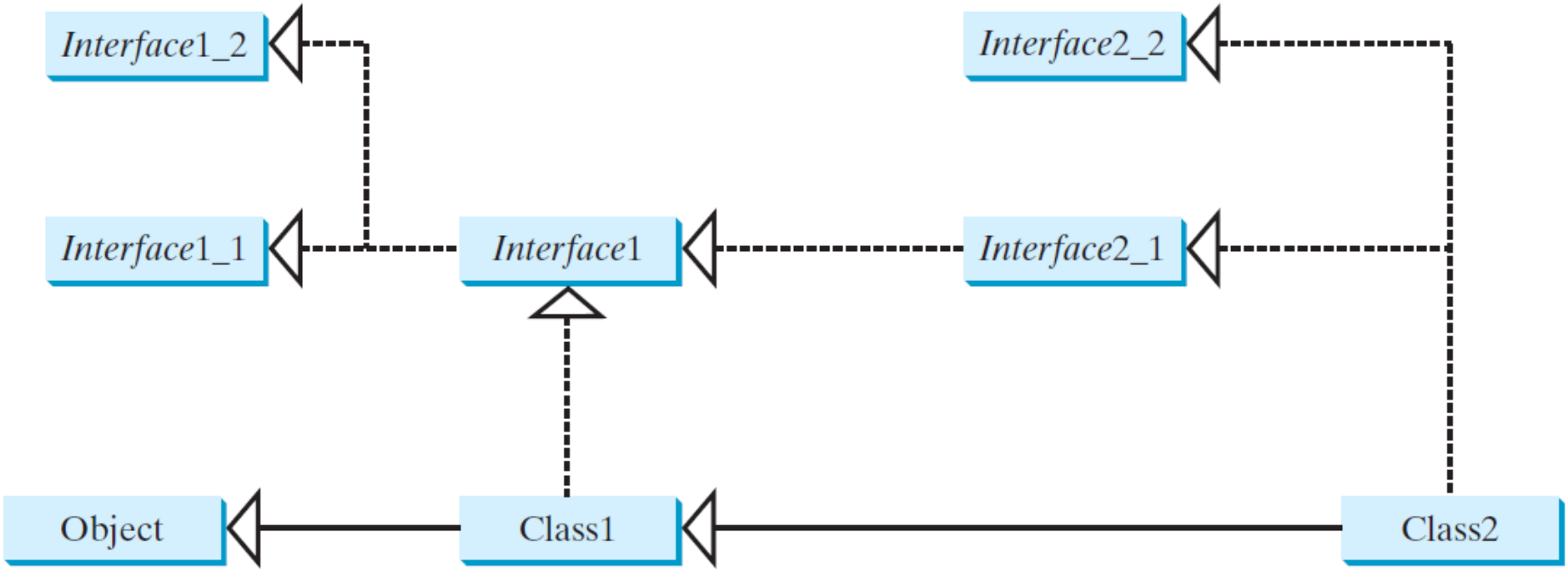
A class can implement several interfaces

```
public class Fish implements Edible, Comparable{...}
```

An interface can extend one or more other interfaces

```
public interface A extends B {...}
```

```
public interface A extends B, C {...}
```



Some rules, cont.

Omitting modifiers in an interface:

- all data fields are **`public static final`** and
- all methods are **`public abstract`** in an interface.

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

Accessing constants:

- A constant defined in an interface can be accessed using the dot operator along with the name of the *interface* or the *implementing class* (or an instance of that class)
- For example:

```
T1.K
```

Exercise

Assume

- **Employee** and **Student** are classes;
- **Comparable** and **Serializable** are interfaces.

Which of these declarations is valid, and why?

a) `class WorkingStudent extends Employee, Student implements Comparable`

b) `class WorkingStudent extends Comparable`

c) `class WorkingStudent extends Employee implements Student`

d) `class WorkingStudent implements Comparable, Serializable`

e) `public interface x extends Comparable`

f) `public interface x extends Student`

g) `Comparable employee1 = new Comparable();`