CSN-103: Fundamentals of Object Oriented Programming

Instructor: Dr. Rahul Thakur

Assistant Professor, Computer Science and Engineering, IIT Roorkee

Interfaces

- Interfaces allow you to fully abstract a class
  - Specify what a class **must** do, but **not** how it does it
- Interfaces are syntactically similar to classes (but not classes)
  - Without instance variables
  - Methods are declared without any body (sure??)
- Any number of classes can use (implement) an **interface**
- One class can implement **any number** of interfaces
- To implement an interface
  - A class must provide the complete set of methods required by the interface

Defining an Interface

- Much like a Class. General form:

```
access interface name {
      type static-final-varname1 = value;
      type static-final-varname2 = value;
      // ...
      return-type method-name1(parameter-list);
      return-type method-name2(parameter-list);
       // ...
}
```

Defining an Interface

- Access is either not used or **public**
  - Not used (default): Can be used within the package
  - public: Interface can be used by any other code
- Methods have no bodies
  - Methods have no default implementation (Not entirely true)
    - They are, essentially, abstract methods
  - Prior to JDK 8, an interface could not define any implementation whatsoever
    - With JDK 8, it is possible to add a ***default implementation*** to an interface method

Defining an Interface

- Variables can be declared inside of interface declarations
- Variables are implicitly **final** and **static**
  - They cannot be changed by the implementing class
  - They must also be initialized
- All **methods** and **variables** are implicitly **public**
  - No need to use public keyword

Implementing Interfaces

- Use the **implements** clause in a class definition, and then create the methods defined by the interface
- General form:

```
class classname implements interface{
        // class-body
}
```

- If a class implements **more than one interface**, the interfaces are separated with a comma

```
class classname implements interface [,interface...]] {
        // class-body
}
```

Inheriting and Implementing

- General Form

```
class classname [extends superclass] [implements interface [,interface...]] {
        // class-body
}
```

- **extends** vs **implements** keyword

Base Class

Interface?

Interface

Interface References

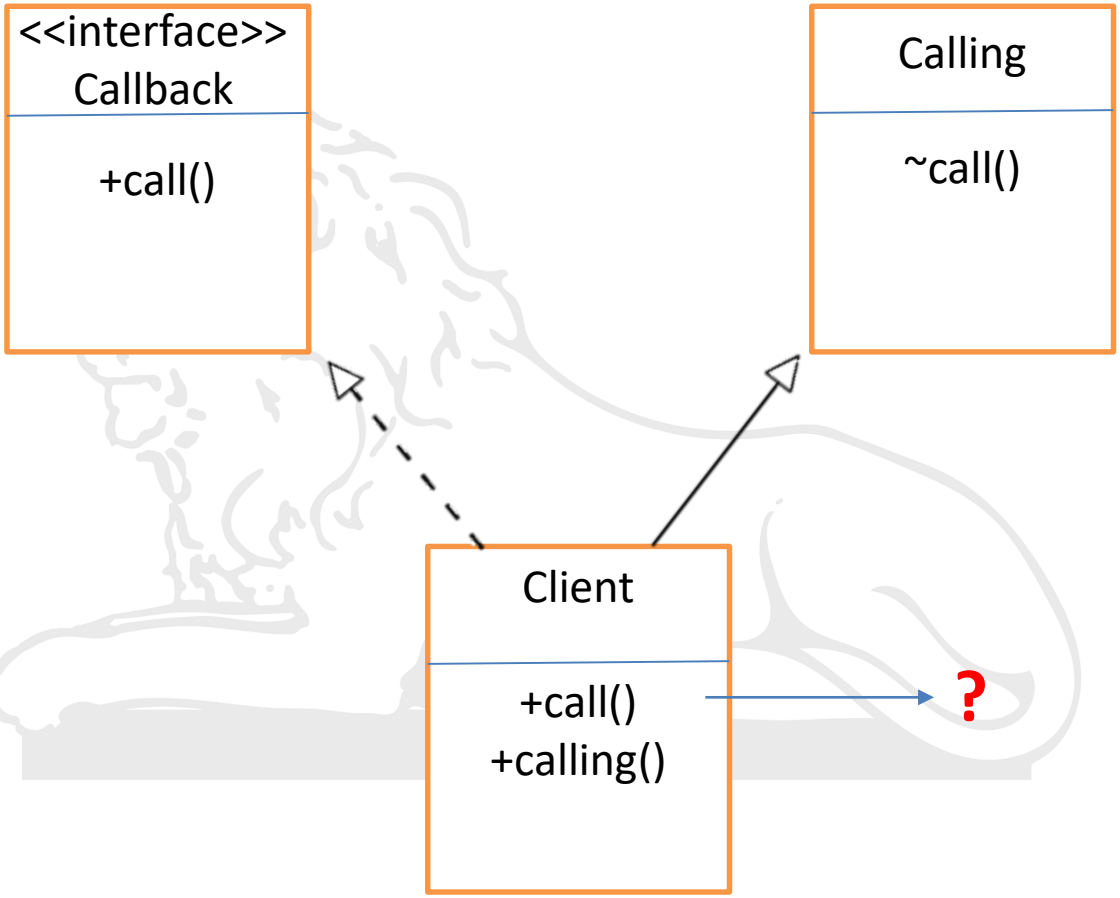
- Declare variables as object references that use an interface rather than a class type
- Any class that implements the declared interface can be referred to by such a variable
- Similar to using a **superclass reference** to access a **subclass object**

```
Callback c = new Client();
```

Interface

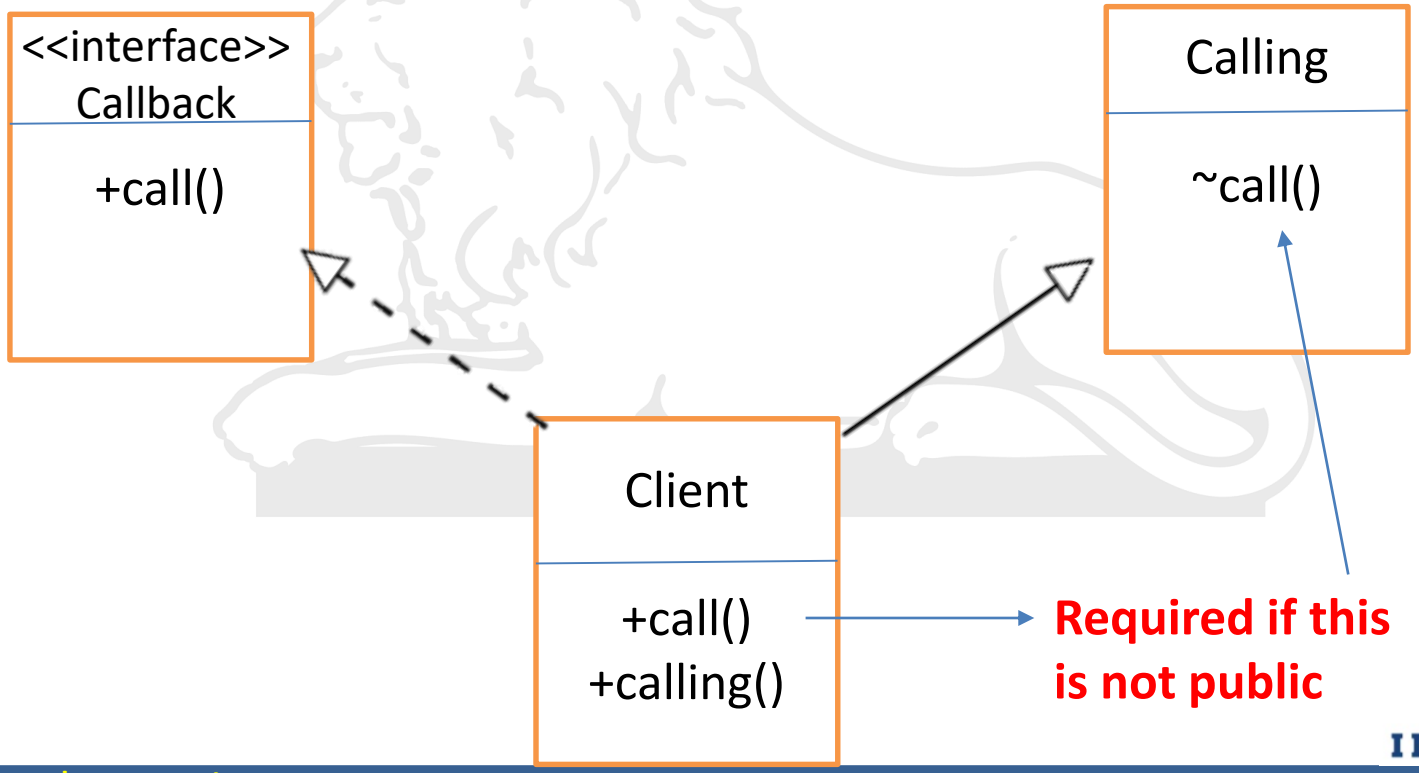
Object of Client Class

Interface and Overriding



Interface and Overriding

- Can not assign weaker access privilege to methods of an interface
- Can use super() to call the method of **superclass**



Partial Implementations

- If a class includes an interface but does not fully implement the methods
  - Then, class must be declared as **abstract**
- Example:

```
interface Callback {
    void callback(int param);
}

abstract class Incomplete implements Callback {
    int a, b;
    void show() {
          System.out.println(a + " " + b);
    }
    //...
}
```

Extending an Interface

- One interface can **inherit** another by use of the keyword **extends**
- The syntax is the same as for inheriting classes
- A class implements an interface that inherits another interface
  - Implementations for all methods are required

```
interface A {
    void meth1();
    void meth2();
}

interface B extends A {
    void meth3();
}
```

```
Class C implements B{
        void meth1(){
        //Body
        }
        void meth2(){
        //Body
        }
        void meth3(){
        //Body
        }
}
```

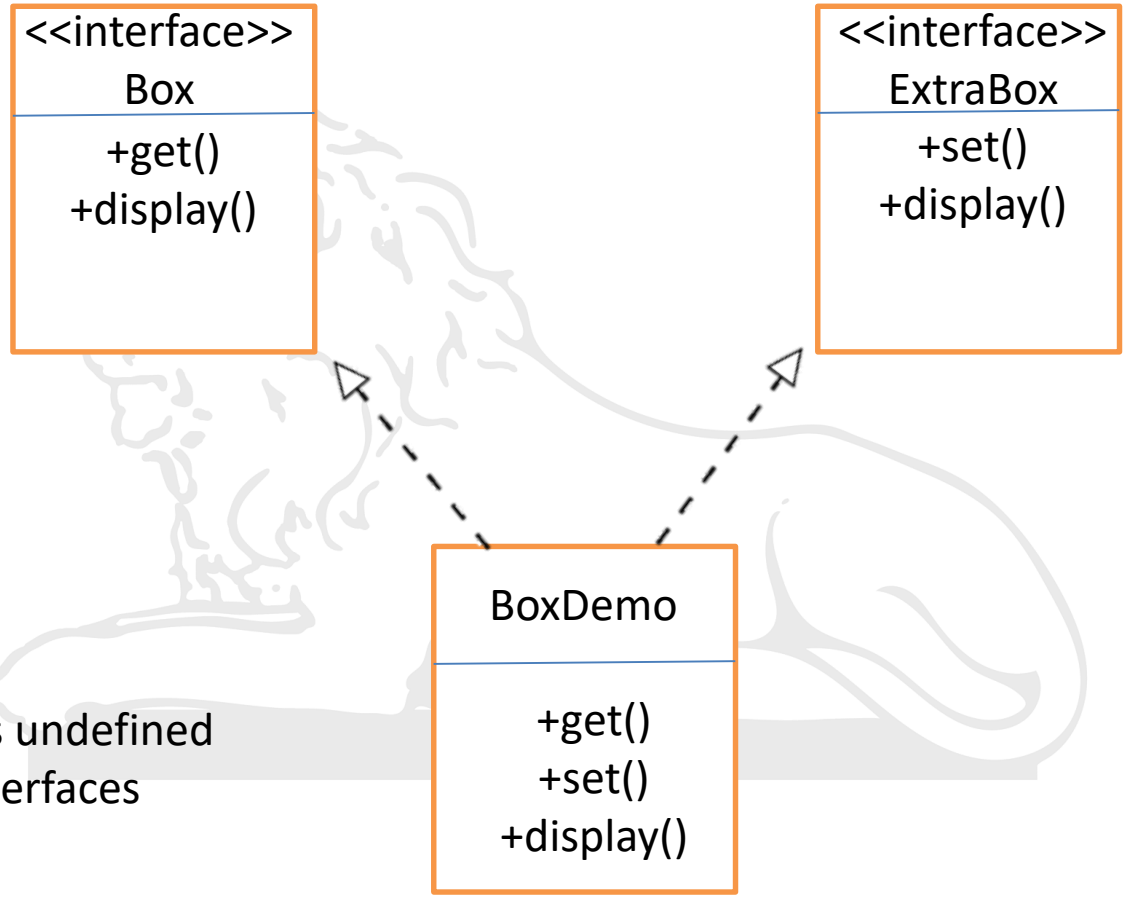
Extends and Implements

- A Class can **extends** a class
- A Class can **implements** interface(s)
- An Interface can **extends** interface(s)
- General Form

class *classname* [extends *superclass*] [implements *interface* [,*interface*...]] {
    // class-body
}

interface *interfacename* [extends *interface* [,*interface*...]] {
    // interface-body
}

Implementing Multiple Interfaces



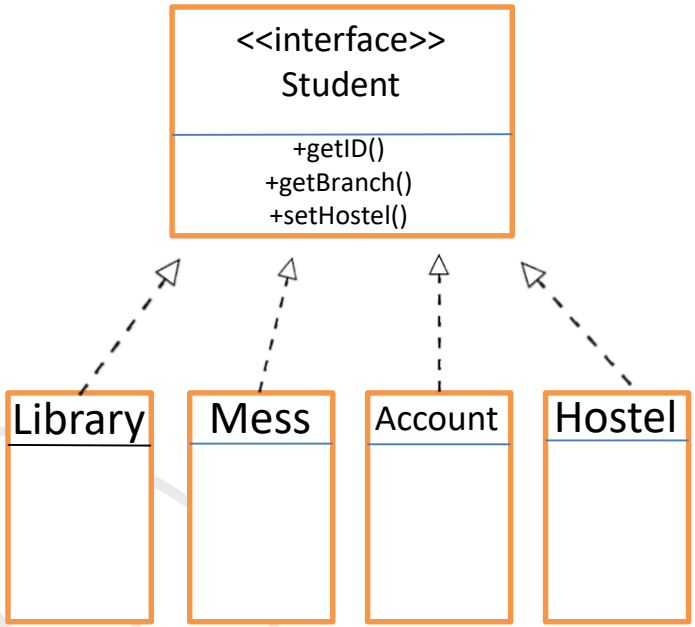
display() is undefined
in both interfaces

Default Interface Methods

- The methods specified by a **traditional** interface were abstract, containing no body
- JDK 8 added a new capability to **interface** called the ***default method***
  - Lets you define a default implementation for an interface method

Why Default Interface Methods?

- Recall that there **must be** implementations for all methods defined by an interface
- Example: A popular, widely used interface such as **Student**
- The addition of an additional method will not cause pre-existing code to **break**



Why Default Interface Methods?

- Another motivation: To specify methods in an interface that are, essentially, optional
- Default methods does not change a key aspect of **interface**: its inability to maintain state information
  - An interface still cannot have instance variables
- It is still not possible to create an instance of an interface by itself

Default method gives you added **flexibility**

Default Method Fundamentals

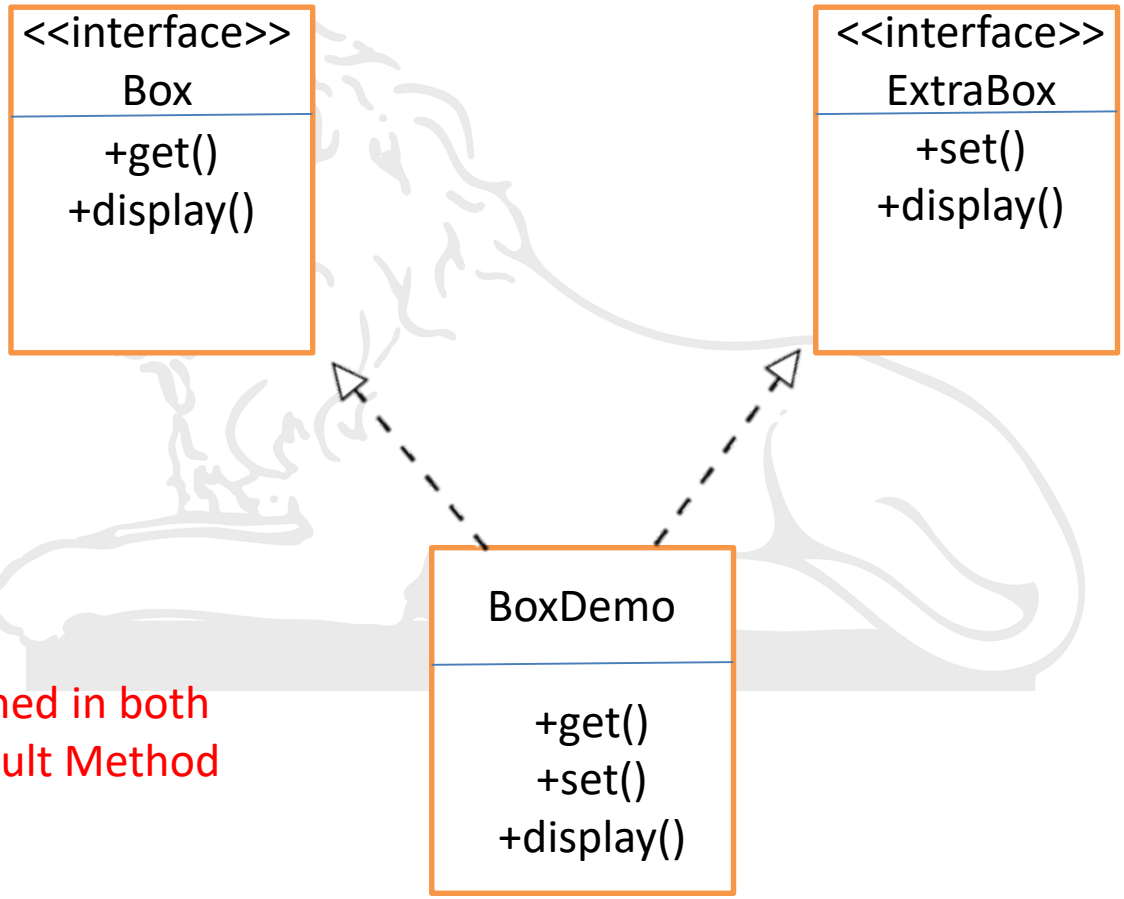
- To define default method → Precede declaration by the keyword **default**

```
public interface MyIF {
      int getNumber();
      default void printString() {
      System.out.println("Default String“);
      }
}
```

- It is not **necessary** to override the printSting()

Multiple Inheritance Issue with Default Method

- Now, an interface can include default methods



display() is defined in both interfaces: Default Method

BoxDemoMultiple.java

static Methods in an Interface

- JDK 8 added another new capability to **interface**
  - Ability to define one or more **static** methods
- **static** methods defined by an interface can be called independently of any object
- No implementation of the **interface** is necessary
- Called by specifying the interface name, followed by a period, followed by the method name
- General Form

InterfaceName.*staticMethodName*

- **static** interface methods are **not inherited** by either an implementing class or a sub-interface