CMSC 132: Object-Oriented Programming II

Interface
Java Interfaces

- A **Java Interface** is a formal way for a class to promise to implement certain methods. We say that a class **implements** an interface if it provides these methods.

- **Interface**:  
  - Is defined by the keyword **interface** (rather than **class**).  
  - It defines **methods** (as many as you like), but does **not** give **method bodies** (the executable statements that make up the method).  
  - All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.  
  - Notice that an **interface is not a class**. You **cannot** create an instance of an interface.
Defining an Interface

- Defining a Java Interface:

  ```java
  public interface Set<E> {
      public void insert(E e);
      public void clear();
      public boolean contains(E o);
      public boolean isEmpty();
      public boolean remove(E o);
      public int size();
  }
  ```
Implementing an Interface

• A class is said to “implement” an interface if it provides definitions for these methods

```java
public class BagSet<E> implements Set<E>{
    ...
}
```

• Now, we may use a BagSet any place that an object of type Set is expected
• A class implementing an interface can implement additional methods
• A class can implement several interfaces
Motivation for Interfaces

• **Two Opposing Goals**, which Java programmers must deal with:
  
  **Strong typing and General-Purpose Functions**

• **Strong Typing**: In strongly typed languages, like Java, the type of every variable must be specified. This makes debugging much easier.

• **General-Purpose Functions**: We would like to write methods that can be applied to many different types. For example, methods for sorting that can work with ints, doubles, Strings, etc. Advantages:
  
  • Less Coding
  • Less likely to have typos
  • Easier maintenance of code

• **The Problem**: Strong typing implies that, for example, to write a sorting function, we need to specify the types of the parameters (int, double, String, etc.). This makes it impossible to write a generic sorting function. It would seem that we need to implement many sorting functions (sortInts(), sortDoubles(), sortStrings(), sortDates(), sortRationals(), …)

• **The Solution**: How can we solve the problem? By using Interfaces!
Java Interfaces

• **How it works:** Suppose you want to write a sorting method for objects of some class X. Sorting requires that you be able to compare the relative values of objects (<, >, <=, >=, ==)
  - You implement a *general-purpose sorting method* using a comparison method (e.g., `compareTo( )`)
  - The user of your sorting function *defines this comparison method* (`compareTo( )`) for objects of class X.
  - Now it is possible to *invoke* your general sorting method on objects of class X

• **To make this work:** Java needs to provide some mechanism for general-purpose functions (like sort) to specify *what behavior they require* from specific classes (like X). This is the purpose of a Java interface.
Comparable Interfaces

- The **Comparable** interface specifies a method called `compareTo` that takes an object as a parameter and returns a negative integer, zero, or a positive integer as the current object is less than, equal to, or greater than the specified object.
- Have we seen classes implementing this interface? Yes!
  - **String**
  - **Integer**
  - **Double**
    - All primitive wrapper classes implement **Comparable**
- By using interfaces a function like `Collections.sort()` can sort an `ArrayList` of objects that implement the `Comparable` interface. For example, an `ArrayList` of Integers, of Strings, etc.
- Can `Collections.sort()` sort an `ArrayList` of your own objects (e.g., `ArrayList` of Cars?) Yes! Just make the Car class implement the **Comparable** interface.
Multiple Inheritance

- There are many situations where a simple class hierarchy is *not adequate* to describe a class’ structure.

- **Example:** Suppose that we have our class hierarchy of *university people* and we also develop a class hierarchy of *athletic people*:

  - **StudentAthlete:** Suppose we want to create an object that inherits all the elements of a *Student* (admission year, GPA) as well as all the elements of an *Athlete* (sport, amateur-status).
Multiple Inheritance

• Can we define a **StudentAthlete** by inheriting all the elements from both **Student** and **Athlete**?

```java
public class StudentAthlete extends Student, extends Athlete {
    ... }
```

• Alas, no. At least not in Java

**Nice try! But not allowed in Java**

• **Multiple Inheritance:**
  • Building a class by extending multiple base classes is called **multiple inheritance**
  • It is a very powerful programming construct, but it has many **subtleties** and **pitfalls**. (E.g., If Athlete and Student both have a **name** instance variable and a **toString( )** method, which one do we inherit?)
  • Java **does not** support multiple inheritance. (Although C++ does.)
    • In Java a class can be **extended** from **only one** base class
    • However, a class can **implement** any **number** of **interfaces**.
Multiple Inheritance with Interfaces

• Java lacks multiple inheritance, but there is an alternative. What public methods do we require of an Athlete object?
  • String getSport( ): Return the athlete’s sport
  • boolean isAmateur( ): Does this athlete have amateur status?
• We can define an interface Athlete that contains these methods:
  
  public interface Athlete {
    public String getSport( );
    public boolean isAmateur( );
  }

• Now, we can define a StudentAthlete that extends Student and implements Athlete
Multiple Inheritance with Interfaces

- StudentAthlete **extends** Student and **implements** Athlete:

```java
public class StudentAthlete extends Student implements Athlete {
    private String mySport;
    private boolean amateur;
    // ... other things omitted
    public String getSport() { return mySport; }
    public boolean isAmateur() { return amateur; }
}
```

- **StudentAthlete** can be used:
  - Anywhere that a **Student object is expected** (because it is derived from Student)
  - Anywhere that an **Athlete object is expected** (because it implements the public interface of Athlete)

- So, we have effectively achieved some of the goals of **multiple inheritance**, by using Java’ single inheritance mechanism
Common Uses of Interfaces

• Interfaces are flexible things and can be used for many purposes in Java:
  • A work-around for Java’s lack of multiple inheritance. (We have just seen this.)
  • Specifying minimal functional requirements for classes (This is its principal purpose.)
  • For defining groups of related symbolic constants. (This is a somewhat unexpected use, but is not uncommon.)
Using Interfaces for Symbolic Constants

• In addition to containing method declarations, interfaces can contain constants, that is, variables that are public final static.

```java
interface OlympicMedal {
    static final String GOLD = "Gold";
    static final String SILVER = "Silver";
    static final String BRONZE = "Bronze";
}
```

• Considered bad practice.
Default Methods

- Java 8 introduces “Default Method”, a new feature
- Add new methods to the interfaces without breaking the existing implementation of these interface.

```java
public interface A {
    public void m1();
    default public void m2 {
        println("default m2");
    }
}

public class B implements A {
    public void m1() {...}
}

B b = new B();
b.m2(); // print “default m2”
```
Abstract classes versus interfaces

- After introducing *Default Method*, it seems that interfaces and abstract classes are same.
- However, they are still different concept in Java 8.
- Abstract class can define constructor. They can have a state associated with them.
- *default method* can be implemented only in the terms of invoking other interface methods, with no reference to a particular implementation's state.
- Both use for different purposes and choosing between two really depends on the scenario context.
Since Java class can implement multiple interfaces and each interface can define *default method* with same method signature, therefore, the inherited methods can conflict with each other.

```java
public interface A {
    default int m1() {
        return 1;
    }
}

public interface B {
    default int m1() {
        return 2;
    }
}

public class C implements A, B {
}
```

This code will fail to compile.
Interface Hierarchies

- Inheritance applies to interfaces, just as it does to classes. When an interface is extended, it inherits all the previous methods.
Quiz 1: True /False

An interface can contain following type of members:

• public, static, final fields (i.e., constants)
• default and static methods with bodies

A. True
B. False
Quiz 1: True /False

- An interface can contain following type of members:
  - public, static, final fields (i.e., constants)
  - default and static methods with bodies

A. True
B. False
Quiz 2: True /False

A class can implement multiple interfaces and many classes can implement the same interface.

A. True
B. False
Quiz 2: True /False

A class can implement multiple interfaces and many classes can implement the same interface.

A. True
B. False
abstract class Demo{
    public int a;
    public Demo(){ a = 10; }
    abstract public void set();
    abstract final public void get();
}

class Test extends Demo{
    public void set(int a){this.a = a;}
    final public void get(){
        System.out.println("a = "+a);
    }
}

public static void main(String[] args){
    Test obj = new Test();
    obj.set(20);
    obj.get();        // A. a = 10
    obj.get();       // B. a = 20
    // C. Compile error
}
abstract class Demo{
    public int a;
    public Demo(){ a = 10; }
    abstract public void set();
    abstract final public void get();
}
class Test extends Demo{
    public void set(int a){this.a = a;}
    final public void get(){
        System.out.println("a = " + a);
    }
}
public static void main(String[] args){
    Test obj = new Test();
    obj.set(20);
    obj.get();
}

Final method can’t be overridden. Thus, an abstract function can’t be final.

A. a = 10  
B. a = 20  
C. Compile error
Can an interface extend another interface?

A. No. Only classes can be extended.
B. No. Interfaces cannot be part of a hierarchy.
C. Yes. Since all interfaces automatically extend Object.
D. Yes.
Can an interface extend another interface?

A. No. Only classes can be extended.
B. No. Interfaces cannot be part of a hierarchy.
C. Yes. Since all interfaces automatically extend Object.
D. Yes.
Can an interface be given the private access modifier?

A. No. Then the interface could never be used.
B. No. Since only private classes could use the interface.
C. Yes. This would make all of its methods and constants private.
D. Yes. This would mean that only classes in the same file could use the interface.
Can an interface be given the private access modifier?

A. No. Then the interface could never be used.
B. No. Since only private classes could use the interface.
C. Yes. This would make all of its methods and constants private.
D. Yes. This would mean that only classes in the same file could use the interface.