CMSC 131
Object-Oriented Programming I

Inheritance V

Dept of Computer Science
University of Maryland College Park

This material is based on material provided by Ben Bederson, Bonnie Dorr, Fawzi Emad, David Mount, Jan Plane

Overview

- Inheritance vs. Composition
- Multiple Inheritance
- Interfaces

Inheritance versus Composition

- **Inheritance** is but one way to create a complex class from another. The other way is to explicitly have an instance variable of the given object type. This is called **composition**

Derive a new class from ObjA.

Common Object:

```
public class ObjA {
    public methodA( ) { … }
}
```

Add ObjA as an instance variable.

Inheritance:

```
public class ObjB extends ObjA {
    …
    // call methodA( );
}
```

Composition:

```
public class ObjB {
    ObjA a;
    // call a.methodA( )
}
```

- **When should I use inheritance vs. Composition?**
  - **ObjB "is a" ObjA**: in this case use **inheritance**
  - **ObjB "has a" ObjA**: in this case use **composition**

Inheritance versus Composition

- **University parking lot permits**: A parking permit object involves a university Person and a lot name ("4", "11", "XX", "Home Depot")

Inheritance:

```
public class Permit extends Person {
    String lotName;

    // ...
}
```

Composition:

```
public class Permit {
    Person p;
    String lotName;
    // ...
}
```

- **Which to use?**
  A parking permit "is a" person? Clearly no
  A parking permit "has a" person? Yes, because a Person is one of the two entities in a a permit object
  So **composition** is the better design choice here
- **Prefer Composition over inheritance**
  When in doubt or when multiple choices available, prefer composition over Inheritance

Review of Overloading and Overriding

- Before discussing interfaces, let's review some elements of method **overloading** and **overriding**

- When **overriding** a method the subclass method prototype must match **exactly** the prototype of the superclass (same name, same return type, same arguments)

- You may change **access specifier** (public, private, protected), but derived classes **cannot decrease the visibility**
  - **Example:** clone() method in Object class

Example: You be the Compiler

```
public class Base {
    protected void someMethod( int x ) { … }
}

public class Derived extends Base {

    public void someMethod( int x ) { … }

    public int someMethod( int x ) { … }

    public void someMethod( double d ) { … }
}
```

Base class

Derived class

Overriding: with increased visibility

Error! duplicate method declaration

Overloading

(the following appears in the same package)

```
Base b = new Base( );
Base d = new Derived( );
Derived e = new Derived( );
b.someMethod( 5 );
d.someMethod( 6 );
d.someMethod( 7.0 );
e.someMethod( 8.0 );
```

calls Base:someMethod(int)

calls Derived:someMethod(int)

Error! Since d is declared Base, this attempts to call the overridden method someMethod(int). But the argument is of the wrong type.

calls Derived:someMethod(double)

Interfaces: Recap

- We introduced the concept of interfaces earlier this semester. Recall:
- **Interface**:
  - Is defined by the keyword **interface** (rather than **class**)
  - It is **abstract**. That is, it defines **methods** (as many as you like), but does **not** give **method bodies** (the executable statements that make up the method)

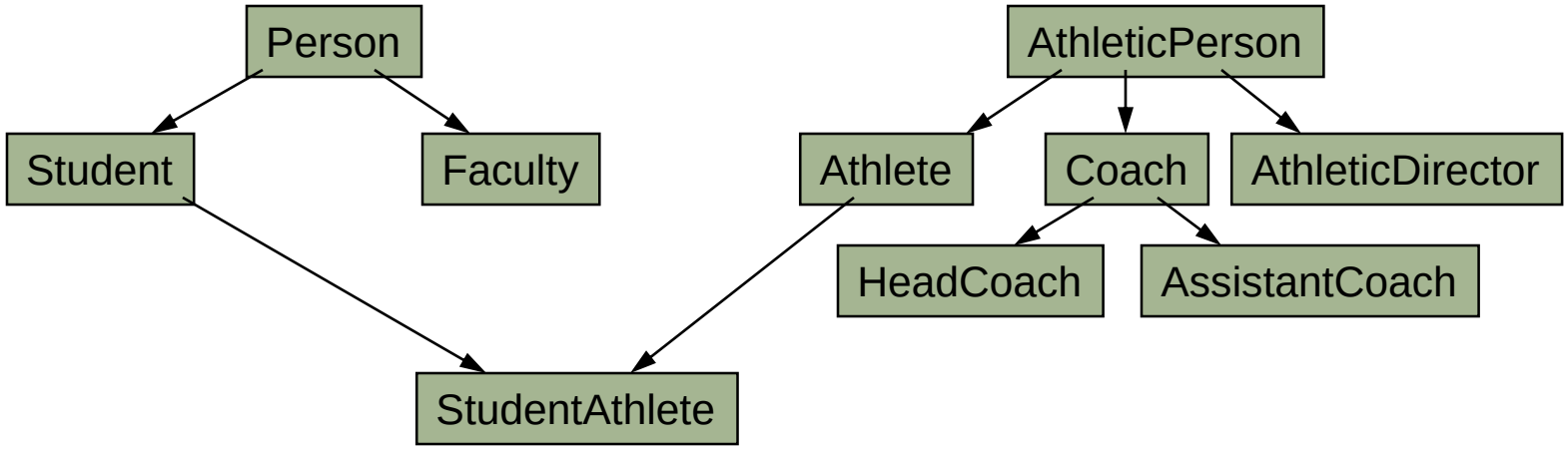
```
public interface Y {
    public void someMethod( int z );
    public int anotherMethod( );
}
```

Methods are declared, but no method bodies are provided.

  - These methods are usually **public**, since they are expected to be part of an object's **public interface**
  - An **interface is not a class**. Because an interface is abstract, you **cannot** create an instance of interface Y using "new Y"

Multiple Inheritance

- Motivation: 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**?

  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**.

"Faking" 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

"Faking" Multiple Inheritance with Interfaces

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

```
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**. Sometimes interfaces are used just for this purpose
- **Example:** IceCreamStore.java

Interface Hierarchies

- Inheritance applies to interfaces, just as it does to classes. When an interface is **extended**, it inherits all the previous methods
- **Example:** InternationalIceCreamStore.java