CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Abstract Classes and Inheritance

Department of Computer Science
University of Maryland, College Park

Motivating Example - Shapes

- Graphics drawing program to create circles, rectangles, etc
  - Define a base class Shape
  - Derive various subclasses for specific shapes
  - Each subclass defines its own method drawMe( )

```
public class Shape {
    public void drawMe( ) { … }       // generic drawing method
}
public class Circle extends Shape {
    public void drawMe( ) { … }       // draws a Circle
}
public class Rectangle extends Shape {
    public void drawMe( ) { … }       // draws a Rectangle
}
```

- If we only need the drawMe() method, could we have used an interface?
- We want to place common methods in base class (in addition to have drawMe())

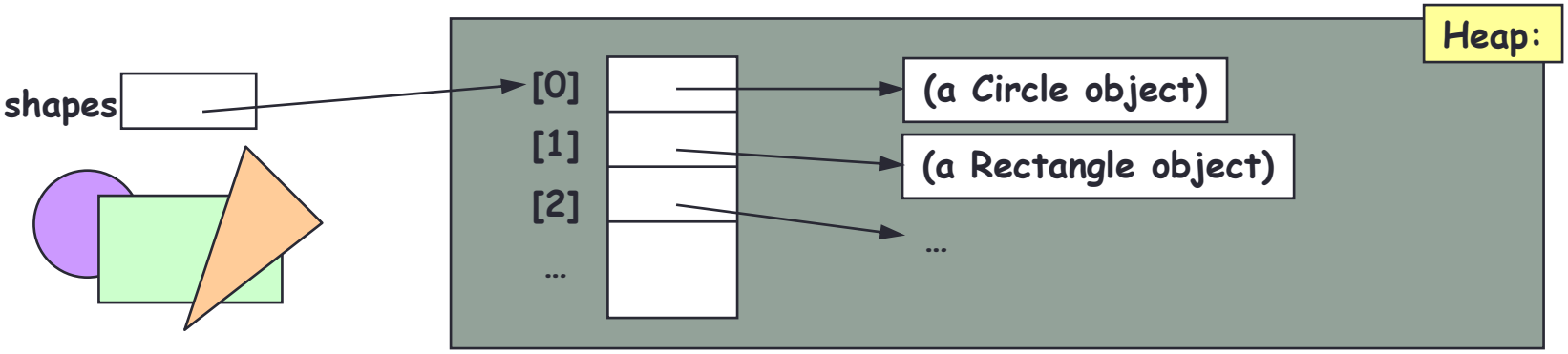
Motivating Example – Shapes

- Implementation
  - Picture consists of array shapes of type Shape[ ]
  - To draw the picture, invoke drawMe( ) for all shapes

```
Shape[ ] shapes = new Shape[…];
shapes[0] = new Circle( … );
shapes[1] = new Rectangle( … );
…
for ( int i = 0; i < shapes.length; i++ )
    shapes[i].drawMe( );
```

Store the shapes to be drawn in an array.

Draws all the shapes. Each call invokes drawMe for the specific shape.



- **Example:** withoutAbstractClass

Motivating Example - Shapes

- **Problem**
  - Shape object does not represent a specific shape, still users can create instances of it (Shape s = new Shape())
  - How to implement Shape's drawMe( ) method?

```
public class Shape {
    void drawMe( ) { … }          // generic drawing method
}
```

- **Possible solutions**
  - Draw some special "undefined shape"
  - Ignore the operation
  - Issue an error message
  - Throw an exception
- **Better solution**
  - Abstract drawMe( ) method, abstract **Shape** class
  - Tells compiler **Shape** is an incomplete class

Modifier - Abstract

- Description
  - Represents generic concept
  - Just a placeholder
  - Leave lower-level details to subclass
- Applied to
  - Methods
  - Classes
- Example

```
abstract class Foo {                    // abstract class
    abstract void bar( );               // abstract method
}
```

- **Example:** withAbstractClass

Abstract Class Summary

- **Abstract Methods**
  - Behaves much like method in interface
  - Give a signature, but no body
  - Includes modifier abstract in method signature
  - Class descendants provide the implementation
  - Abstract methods cannot be final
    - Since must be overridden by descendent class (final would prevent this)
  - A non-abstract method of an abstract class can call abstract methods of the class
- **Abstract Class**
  - Required if class contains any abstract method
  - Includes modifier abstract in the class heading

    public **abstract** class Shape { … }
  - An abstract class is incomplete
    - Cannot be created using "new" → Shape s = new Shape(…); // Illegal!
    - But you can create concrete shapes (Circle, Rectangle) and assign them to variables of type Shape → Shape s = new Circle(…);

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;

     // …
}
```

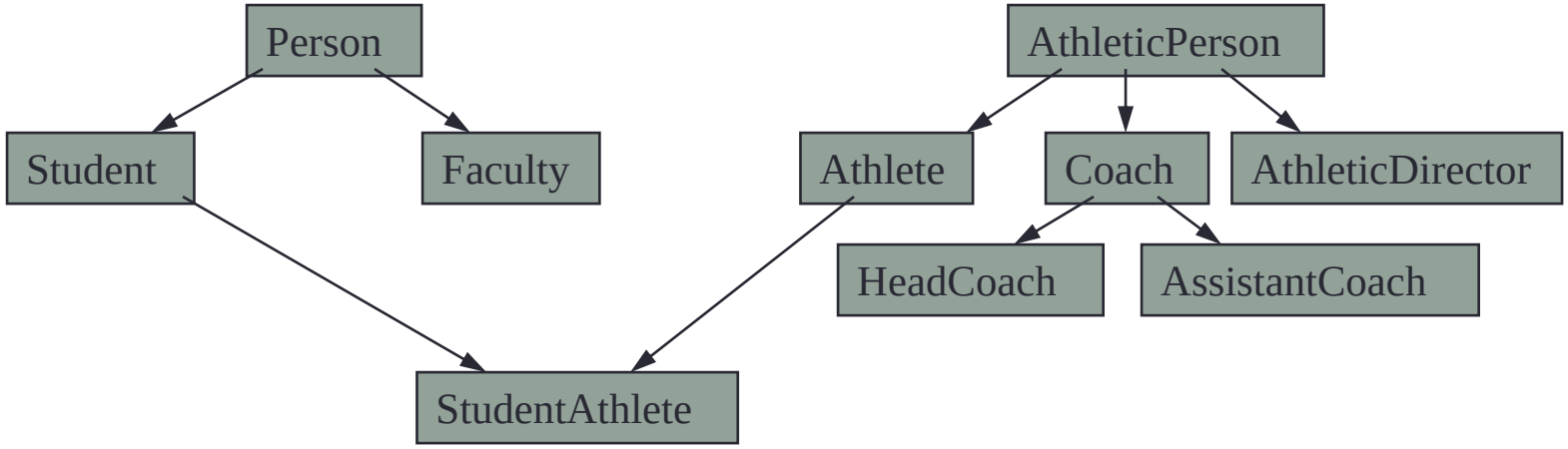
Composition:

```
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 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**

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 extend only one 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)

Interface Hierarchies

- Inheritance applies to interfaces, just as it does to classes. When an interface is **extended**, it inherits all the previous methods
- **Example:** IceCreamStore.java, TerpStore.java, InternationalIceCreamStore.java (inherits from IceCreamStore.java), IceCreamChamp.java (implements InternationalIceCreamStore), Driver.java

Review of Overloading and Overriding

- Let's review some elements of method **overloading** and **overriding**
- **Method's signature –** includes only the name, and parameters
- **Method's prototype** – first line of the method definition with a semicolon at the end
- When **overriding** a method, the subclass method signature must match **exactly** the signature of the superclass (same name, same arguments)
- You may change **access specifier** (public, private, protected), but derived classes **cannot decrease the visibility**
  - **Example:** clone() method in Object class
    - By default defined **protected**, but when we override it we define it as **public**
- Example of **overloading**: max/min methods in Math class
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html

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

When analyzing the following, first consider whether a statement compiles. All the following are 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)

Disabling Overriding with "final"

- We can disable overriding by declaring a method to be "**final**"
- Sometimes you do not want to allow method overriding

  Correctness: Your method only makes sense when applied to the base class. Redefining it for a derived class might break things

  Efficiency: Late binding is less efficient than early binding. You know that no subclass will redefine your method. You can force early binding by disabling overriding
- **Example**: The class **Object** defines the following method:

  getClass( ): returns a description of a class. You can test whether two objects x and y are of the same class with:

  if (x.getClass( ) == y.getClass( )) …

  This is a very useful function. But clearly, we do not want arbitrary classes screwing around with it. The **getClass()** method is a final
- **Example:** getArea() final method in **withAbstractClass.Circle**

Disabling Overriding with "final"

- **final**: Has different meanings, depending on context:
  - Define **symbolic constants**:

```
public static final int MAX_BUFFER_SIZE = 1000;
```

  - Indicate that a method **cannot be overridden by derived classes**

```
public class Parent {
    public final void someMethod( ) { … }
}
public class Child extends Parent {
    public void someMethod( ) { … }
}
```

Subclasses cannot override this method

Illegal! someMethod is final in base class.

- A class can be defined as final what will not allow the class to be extended. For example, public **final** class Circle extends Shape will not allow us to define a **SuperCircle** class that extends **Circle**
- A final class cannot be extended
  - String class is an example of a final class
    - Too important for others to change the behavior associated with String methods