CMSC 150
INTRODUCTION TO COMPUTING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO PROGRAMMING IN JAVA: AN INTERDISCIPLINARY APPROACH, SEDGEWICK AND WAYNE (PEARSON ADDISON-WESLEY 2007)

LECTURE 11

• POLYMORPHISM
• ABSTRACT CLASSES
• INTERFACES
REVIEW
DATA TYPES AND OBJECT-ORIENTED PROGRAMMING

• **Data type. Object.** Set of values and operations on those values.

• **Object-oriented Programming** – design principle for large programs
  • **Composition/Abstraction** – Modeling objects (HAS-A relationship)
  • **Encapsulation** – combining data and operations (methods); data hiding from misuse (private vs public)
  • **Inheritance** – Types and sub-types (IS-A relationship)
  • **Polymorphism** – Abstract types that can act as other types (for algorithm design)
EXAMPLE
SHAPES

- Recall our shape hierarchy
- Shape will have the functions
  - `double area();`
  - `double perimeter();`
- Specifics are defined in the sub classes

```
Shape
  • Color

  Circle
    • Radius

  Rectangle
    • Width, Height
```
POLYMORPHISM

• Wikipedia – “the provision of a single interface to entities of different types”
• “one name, many forms”
• **Polymorphism** realistically implies that a variable of a superclass can refer to a value of a subclass

```java
Shape circle = new Circle(5, Color.red);
System.out.println(circle.area());
```
WHY WOULD YOU EVER DO THIS?

• Allow types to be defined at runtime, instead of at compile time:

```java
int n = 5;
if (n > 0) {
    System.out.println("positive");
}
else {
    System.out.println("negative");
}
```

```java
Scanner s = new Scanner(System.in);
Shape shape = null;
String tag = s.next();
if (tag.equals("Circle")) {
    //user wants a circle
    double r = s.nextDouble();
    shape = new Circle(r, Color.red);
}
else if (tag.equals("Rectangle")) {
    //User wants a rectangle
    double w = s.nextDouble(), h = s.nextDouble();
    shape = new Rectangle(w, h, Color.red);
}
System.out.println("Area: " + shape.area());
//works no matter what!
```
WHY WOULD YOU EVER DO THIS?

- Arrays can only store one type

1. `Circle[] circles;` //all circles
2. `Rectangle[] rects;` //all rectangles
3. `Shape[] shapes;` //depends on subtypes! Can have some circles and some rectangles.
WHY WOULD YOU EVER DO THIS?

• Lets say we have an array of Shape shapes then we can do something like:

1. `double total = 0;`
2. `for (int i = 0; i < shapes.length; ++i)`
3. `total += shapes[i].area(); //Uses specific instance’s subtype’s function`
4. `return total;`
DYNAMIC BINDING

• When defining a variable of a super type as a sub type, e.g.,
  
  \[
  \text{Shape } s = \text{ new Circle}(5, \text{ Color.red});
  \]

  • Shape is the \text{declared type}
  
  • Circle is the \text{actual type}
  
  • \text{Dynamic binding} relates the correct implementation of the functions to the variable
  
  • The declared type says what functions and public entities can be accessed

    • Note that by declaring \(s\) as \text{Shape}, all of the additional public API functions/data cannot be accessed, e.g., \text{getRadius()}. Lucky for us though…
TYPE CASTING

• Can use casting to get back to the actual type:
  
  ```java
  Shape s = new Circle(5, Color.red);
  Circle c = (Circle)s;  //Only the pointer is copied
  c.specificFunctionInCircleOnly();
  ```

• Casting to a subclass is referred to as **downcasting** and must be done explicitly

• Casting to a superclass is referred to as **upcasting** and will be done implicitly

• Determining if an instance can be downcast is often necessary. Can use the `instanceof` keyword
ABSTRACT CLASSES

• In modeling, sometimes we don’t want to allow types to be defined:
  Shape s = new Shape(Color.red); //Makes no sense. What is s really?

• We can use abstract classes to facilitate this to provide better protection to other software developers on our team. Also specified interface (API) requirements of subtypes.

1. public abstract class Shape {
2.     //Abstract here disbars the code above.
3.     //No “new” is allowed on this type.
4.     protected Shape(Color c) {...} //Constructor is protected because
5.         //nothing but subtypes will access it
6.     ...
7.     public abstract double area(); //If a function is abstract no
8.         //definition needs to be provided
9.     public abstract double perimeter(); //Also subtypes are now required
10.    //to define them!
SOME INTERESTING POINTS ON ABSTRACT

• An abstract method cannot be contained in a non abstract class

• If a subclass of an abstract superclass does not implement all of the abstract methods, then it must also be declared as abstract

• Cannot use new on an abstract type, but constructors can be defined (for use with super). Also can still use the abstract type for polymorphism!

• An abstract class does not require abstract methods

• A subclass can be abstract even if the superclass is concrete (non abstract)
INTERFACES

• An interface is a class-like construct that contains only constants and abstract methods (almost like a purely abstract class).

1. public interface AreaComputation {
   // Note “interface”
   2.       // not “class”
   3.     public static final double PI = Math.PI;
   4.     public abstract area();
   5. }

INTERFACES

• Cannot have constructors

• All variables must be **public static final**

• All methods must be **public abstract**

• Useful for writing algorithms for searching or sorting (these need comparison), i.e., Comparable things (any object “implementing” the Comparable interface)

• Used to support multiple inheritance
INTERFACES

• To inherit an interface:
  
  ```java
  public class Shape implements AreaComputation,
  PerimeterComputation {
  ...
  }
  ```

• Implementing an interface requires implementation of all of the abstract methods, or declaring as an abstract class.

• Interfaces commonly used as a weaker is-a relationship, specifically is-kind-of referring to possessing certain properties only

• Oddly, interfaces can “extend” other interfaces
SUMMARY OF OOP

• OOP is a methodology to model things in our world and their interactions
  • Used for solving problems
  • Used in creating useful applications

• Do not think this is the end of the story…
  • We only went over the core principles of OOP
  • There are more advanced programming techniques
  • There are many differences in OOP between languages