Topic 7
Interfaces and Abstract Classes

“I prefer Agassiz in the abstract, rather than in the concrete.”
Interfaces
Multiple Inheritance

- The are classes where the “is-a” test is true for more than one other class
  - a graduate teaching assistant is a graduate students
  - a graduate teaching assistant is a faculty member

- Java requires all classes to inherit from exactly one other class
  - does not allow multiple inheritance
  - some object oriented languages do
Problems with Multiple Inheritance

- Suppose multiple inheritance was allowed
  ```java
  public class GradTA extends Faculty, GradStudent
  ```
- Suppose Faculty overrides toString and that GradStudent overrides toString as well
  ```java
  GradTA ta1 = new GradTA();
  System.out.println( ta1.toString() );
  ```
- What is the problem
- Certainly possible to overcome the problem
  – provide access to both (scope resolution in C++)
  – require GradTA to pick a version of toString or override it itself (Eiffel)
Interfaces – Not quite Multiple Inheritance

- Java does not allow multiple inheritance
  – syntax headaches not worth the benefits
- Java has a mechanism to allow specification of a data type with NO implementation
  – interfaces
- Pure Design
  – allow a form of multiple inheritance without the possibility of conflicting implementations
A List Interface

- What if we wanted to specify the operations for a List, but no implementation?
- Allow for multiple, different implementations.
- Provides a way of creating *abstractions*.
  - a central idea of computer science and programming.
  - specify "what" without specifying "how"
  - "Abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time. "

Abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time.
public interface List{
    public void add(Object val);
    public int size();
    public Object get(int location);
    public void insert(int location, Object val);
    public void addAll(List other);
    public Object remove(int location);
}

Interfaces

- All methods in interfaces are public and abstract
  - can leave off those modifiers in method headers
- No constructors
- No instance variables
- can have class constants
  
  ```
  public static final int DEFAULT_SIDES = 6
  ```
Implementing Interfaces

- A class inherits (extends) exactly one other class, but ...
- A class can *implement* as many interfaces as it likes
  ```java
  public class ArrayList implements List
  ```
- A class that implements an interface must provide implementations of all method declared in the interface or the class must be abstract
- Interfaces can extend other interfaces
Why interfaces?

- Interfaces allow the creation of *abstract data types*
  - "A set of data values and associated operations that are precisely specified independent of any particular implementation."
  - multiple implementations allowed

- Interfaces allow a class to be specified without worrying about the implementation
  - do design first
  - What will this data type do?
  - Don’t worry about implementation until design is done.
  - separation of concerns

- allow a form of multiple inheritance
The Comparable Interface

- The Java Standard Library contains a number of interfaces – names are italicized in the class listing
- One of the most important interfaces is the Comparable interface
Comparable Interface version 1.4

```java
package java.lang

public interface Comparable
{
    public int compareTo( Object other );
}
```

- compareTo should return an int <0 if the calling object is less than the parameter, 0 if they are equal, and an int >0 if the calling object is greater than the parameter.
Implementing Comparable

- Any class that has a *natural ordering* of its objects (that is objects of that type can be sorted based on some internal attribute) should implement the Comparable interface.

- Back to the `ClosedShape` example

- Suppose we want to be able to sort `ClosedShapes` and it is to be based on area.
Example `compareTo`

- Suppose we have a class to model playing cards
  - Ace of Spades, King of Hearts, Two of Clubs
- Each card has a suit and a value, represented by ints
- This version of `compareTo` will compare values first and then break ties with suits
**compareTo in a Card class**

```java
public class Card implements Comparable {
    public int compareTo(Object otherObject) {
        Card other = (Card) otherObject;
        int result = this.myRank - other.myRank;
        if (result == 0) {
            result = this.mySuit - other.mySuit;
        }
        return result;
    }
    // other methods not shown
}
```

Assume ints for ranks (2, 3, 4, 5, 6,...) and suits (0 is clubs, 1 is diamonds, 2 is hearts, 3 is spades).
Interfaces and Polymorphism

- Interfaces may be used as the data type for object variables
- Can’t simply create objects of that type
- Can refer to any objects that implement the interface or descendants

Assume `Card implements Comparable`

```java
Card c = new Card();
Comparable comp1 = new Card();
Comparable comp2 = c;
```
Polymorphism Again!

What can this Sort?

```java
public static void SelSort(Comparable[] list)
{
    Comparable temp;
    int smallest;
    for(int i = 0; i < list.length - 1; i++)
    {
        small = i;
        for(int j = i + 1; j < list.length; j++)
        {
            if( list[j].compareTo(list[small]) < 0)
            {
                small = j;
            }
        }
        temp = list[i];
        list[i] = list[small];
        list[small] = temp;
    }
}
```
Abstract Classes

Part Class, part Interface
Back to the ClosedShape Example

- One behavior we might want in ClosedShapes is a way to get the area problem: How do I get the area of something that is “just a ClosedShape”? 
The ClosedShape class

```java
public class ClosedShape {
    private double myX;
    private double myY;

    public double getArea() {
        // Hmmmm?!?!?
    }

    // Other methods not shown
}
```

Doesn’t seem like we have enough information to get the area if all we know is it is a ClosedShape.
Options

1. Just leave it for the sub classes.
   ▸ Have each sub class define `getArea()` if they want to.

2. Define `getArea()` in `ClosedShape` and simply return 0.
   ▸ Sub classes can override the method with more meaningful behavior.
Leave it to the Sub - Classes

// no getArea() in ClosedShape

public void printAreas(ClosedShape[] shapes) {
    for (ClosedShape s : shapes) {
        System.out.println( s.getArea() );
    }
}

ClosedShape[] shapes = new ClosedShape[2];
shapes[0] = new Rectangle(1, 2, 3, 4);
shapes[1] = new Circle(1, 2, 3);
printAreas( shapes );

Will the above code compile?

How does the compiler determine if a method call is allowed?
Fix by Casting

// no getArea() in ClosedShape

public void printAreas(ClosedShape[] shapes)
{
    for( ClosedShape s : shapes )
    {
        if( s instanceof Rectangle )
            System.out.println( ((Rectangle)s).getArea() );
        else if( s instanceof Circle )
            System.out.println( ((Circle)s).getArea() );
    }
}

ClosedShape[] shapes = new ClosedShape[2];
shapes[0] = new Rectangle(1, 2, 3, 4);
shapes[1] = new Circle(1, 2, 3);
printAreas( shapes );

What happens as we add more sub classes of ClosedShape?

What happens if one of the objects is just a ClosedShape?
Fix with Dummy Method

// getArea() in ClosedShape returns 0

class ClosedShape
{
    public int getArea()
    {
        return 0;
    }
}

public class DummyMethod
{
    public static void main(String[] args)
    {
        ClosedShape[] shapes = new ClosedShape[2];
        shapes[0] = new Rectangle(1, 2, 3, 4);
        shapes[1] = new Circle(1, 2, 3);
        printAreas(shapes);
    }

    public static void printAreas(ClosedShape[] shapes)
    {
        for( ClosedShape s : shapes )
        {
            System.out.println( s.getArea() );
        }
    }

    ClosedShape[] shapes = new ClosedShape[2];
    shapes[0] = new Rectangle(1, 2, 3, 4);
    shapes[1] = new Circle(1, 2, 3);
    printAreas( shapes );

    What happens if sub classes don’t override getArea()?

    Does that make sense?
}
A Better Fix

- We know we want to be able to find the area of objects that are instances of `ClosedShape`.
- The problem is we don’t know how to do that if all we know is it a `ClosedShape`.
- Make `getArea` an abstract method.
- Java keyword.
Making getArea Abstract

```java
public class ClosedShape {
    private double myX;
    private double myY;

    public abstract double getArea();
    // I know I want it.
    // Just don’t know how, yet...
}

// Other methods not shown
```

Methods that are declared abstract have no body an undefined behavior.

All methods in an interface are abstract.
Problems with Abstract Methods

Given `getArea()` is now an abstract method what is wrong with the following code?

```java
ClosedShape s = new ClosedShape();
System.out.println( s.getArea() );
```
Undefined Behavior = Bad

- Not good to have undefined behaviors
- If a class has 1 or more abstract methods, the class must also be declared abstract.
  - version of `ClosedShape` shown would cause a compile error
- Even if a class has zero abstract methods a programmer can still choose to make it abstract
  - if it models some abstract thing
  - is there anything that is just a “Mammal”?
Abstract Classes

```java
public abstract class ClosedShape
{
    private double myX;
    private double myY;

    public abstract double getArea();
    // I know I want it.
    // Just don’t know how, yet...

} // Other methods not shown

if a class is abstract the compiler will not allow constructors of that class to be called
ClosedShape s = new ClosedShape(1,2);
//syntax error
```
Abstract Classes

- In other words you can’t create instances of objects where the lowest or most specific class type is an abstract class.
- Prevents having an object with an undefined behavior.
- Why would you still want to have constructors in an abstract class?
- Object variables of classes that are abstract types may still be declared:
  ```java
  ClosedShape s; //okay
  ```
Sub Classes of Abstract Classes

- Classes that extend an abstract class must provided a working version of any abstract methods from the parent class
  - or they must be declared to be abstract as well
  - could still decide to keep a class abstract regardless of status of abstract methods
Implementing getArea()

public class Rectangle extends ClosedShape
{
    private double myWidth;
    private double myHeight;

    public double getArea()
    {
        return myWidth * myHeight;
    }

    // other methods not shown
}

public class Square extends Rectangle
{
    public Square()
    {
    }

    public Square(double side)
    {
        super(side, side);
    }

    public Square(double x, double y, double side)
    {
        super(side, side, x, y);
    }
}
public class Circle extends ClosedShape
{
    double dMyRadius;

    public Circle()
    {
        super(0,0);
    }

    public Circle(double radius)
    {
        super(0,0);
        dMyRadius = radius;
    }

    public Circle(double x, double y, double radius)
    {
        super(x,y);
        dMyRadius = radius;
    }

    public double getArea()
    {
        return Math.PI * dMyRadius * dMyRadius;
    }

    public String toString()
    {
        return super.toString() + " radius: " + dMyRadius;
    }
}
Polymorphism in Action

```java
public class UsesShapes {
    public static void go() {
        ClosedShape[] sList = new ClosedShape[10];
        double a, b, c, d;
        int x;
        for (int i = 0; i < 10; i++) {
            a = Math.random() * 100;
            b = Math.random() * 100;
            c = Math.random() * 100;
            d = Math.random() * 100;
            x = (int)(Math.random() * 3);
            if (x == 0)
                sList[i] = new Rectangle(a, b, c, d);
            else if (x == 1)
                sList[i] = new Square(a, c, d);
            else
                sList[i] = new Circle(a, c, d);
        }
        double total = 0.0;
        for (int i = 0; i < 10; i++)
            total += sList[i].getArea();
        System.out.println(sList[i]);
    }
}
```
The Kicker

- We want to expand our pallet of shapes
- Triangle could also be a sub class of ClosedShape.
  - it would *inherit* from ClosedShape
    ```java
    public double getArea()
    { return 0.5 * dMyWidth * dMyHeight;}
    ```
- What changes do we have to make to the code on the previous slide for totaling area so it will now handle Triangles as well?
- Inheritance is can be described as new code using old code.
- **Polymorphism can be described as old code using new code.**
public abstract class ClosedShape implements Comparable
{
  private double myX;
  private double myY;

  public abstract double getArea();

  public int compareTo(Object other)
  {
    int result;
    ClosedShape otherShape = (ClosedShape)other;
    double diff = getArea() - otherShape.getArea();
    if( diff == 0 )
      result = 0;
    else if( diff < 0 )
      result = -1;
    else
      result = 1;
    return result
  }
}
About ClosedShapes compareTo

- don’t have to return -1, 1.
  - Any int less than 0 or int greater than 0 based on 2 objects
- the compareTo method makes use of the getArea() method which is abstract in ClosedShape
  - how is that possible?