CSCI 261
Computer Science II
Recap

- Polymorphic variables and methods:
  - Variable declared more generically can store specialized objects
  - Polymorphic method calls are resolved at runtime

- For instance,
  - Cars and Trucks are both specialized Vehicles
  - But cars go "beep beep" and trucks go "crank crank"

Vehicle v; //v is a polymorphic variable

v = new Car(); //v can store a Car, because Car is a subtype of Vehicle
v.go(); //beep beep

v = new Truck(); //v can also be a Truck, because Truck is a subtype of Vehicle
v.go(); //crank crank
Recap (Cont.)

- Polymorphism is good
  - A savior for code maintenance
  - Avoids code duplication
  - Simplifies code in the client class
  - Makes it easier to add new subclasses in the future
    - e.g., Goats, Turtles, ...
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
New Example: Shapes

- Write a `ShapesManager` class to manage various shapes
  - All shapes have some things in common
    - Location, color, maybe others
    - Area, but equation depends on the shape...
  - Rectangle
    - Has a width and length
    - Area: width * length
  - Circle
    - Has a radius
    - Area: `Math.PI * radius * radius`
  - Lots more...
Start with Circle

class Circle {
    private int x;
    private int y;
    private double radius;

    public Circle(int x, int y, double r) {
        this.x = x;
        this.y = y;
        this.radius = r;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

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

    //possibly more methods omitted
}
A ShapesManager that Works with Circles

```java
import java.util.ArrayList;

public class ShapesManager {
    private ArrayList<Circle> shapes;

    public ShapesManager() {
        shapes = new ArrayList<Circle>();
    }

    public Circle getShape(int i) {
        return shapes.get(i);
    }

    public void addShape(Circle c) {
        return shapes.add(c);
    }

    public double getAverageArea() {
        double area = 0;
        for (int i = 0; i < shapes.size(); i++) {
            area += shapes.get(i).getArea();
        }
        return area/shapes.size();
    }
}
```
public class Triangle {
    private int x;
    private int y;
    private double base;
    private double height;

    public Triangle(int x, int y, double b, double h) {
        this.x = x;
        this.y = y;
        this.base = b;
        this.height = h;
    }

    public int getX() { return x; }
}

public int getY() { return y; }

public double getArea() {
    return (base * height)/2.0;
}

//possibly more methods omitted
Want ShapesManager to Work with All Shapes

- Extract common fields and methods
  - Why isn't `getArea()` intersected?
  - [Let's create the Shape superclass]
Want ShapesManager to Work with All Shapes

- **Yay our ShapesManager is working now?**
  - Err... nope! It won't compile * BEEN THROUGH THIS ALL LAST LECTURE *

- **Let's see the code for ShapesManager**
import java.util.ArrayList;

public class ShapesManager {
    protected ArrayList<Shape> shapes;

    public ShapesManager() {
        shapes = new ArrayList<Shape>();
    }

    public Shape getShape(int i) {
        return shapes.get(i);
    }

    public void addShape(Shape s) {
        shapes.add(s);
    }

    public double getAverageArea() {
        double area = 0;
        for (int i = 0; i < shapes.size(); i++) {
            area += shapes.get(i).getArea();
        }
        return area/shapes.size();
    }
}
Okay, we *did* need to have common method signatures in `Shape`

- `getArea()`, and possibly others
  - But they all have different implementations in Circle, Triangle, *etc.*
- Seems silly (and unsettling) to have empty methods
  - Also, there's no guarantee a subclass would override it!
    - And that is a concern. What if they forget?

```java
public class Shape {
    protected int x;
    protected int y;

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {  
        return x;
    }

    public int getY() {  
        return y;
    }

    public double getArea() {
        //this is just here so client classes can compile  
        //subclasses will override and provide implementation
        return Double.NaN;
    }
}
```
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
Specifying a Contract

- What we need:
  - A way to specify a contract that certain methods must be implemented in a subclass.
  - Mechanism: Use abstract classes and methods

- Abstract methods
  - Example: `public abstract void foo(int x, String y);`
  - Never has a code-body (implementation done in subclass)
  - Can only exist within an abstract class
  - Contract: Subclasses must override abstract methods with actual implementations
Specifying a Contract (Cont.)

- **Abstract Classes**
  - Example: `public abstract class Classname { ... }`

- **Characteristics**
  - Can contain private/protected/public fields
    - These will be inherited by any subclass
  - Abstract classes may contain constructors but *cannot* be instantiated!
    - *(Then what's the point of having constructors?)*
  - Can contain methods with implementation (concrete) or without (abstract)
    - Concrete methods will be inherited by any subclass
    - Abstract methods must be overridden by any subclass
The only use-case for abstract classes is for subclasses to extend it.

Shape s = new Shape(0,0); //compiler error
A Concrete Circle Class

```java
public class Circle extends Shape {
    protected double radius;

    public Circle(int x, int y, double r) {
        super(x, y);
        this.radius = r;
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }

    //possibly more Circle methods omitted
}
```
Compiler Ensures Contract Is Bound

- Contract states a subclass extending an abstract class must
  - What if we didn't implement *an* abstract method?

```java
public class Circle extends Shape {
    protected double radius;

    public Circle(int x, int y, double r) {
        super(x, y);
        this.radius = r;
    }
}
```

**Contract breached! Results in:**
Compiler error: The type Circle must implement the inherited abstract method: getArea()
Concrete Triangle Class

```java
public class Triangle extends Shape {
    protected double base;
    protected double height;

    public Triangle(int x, int y, double b, double h) {
        super(x, y);
        this.base = b;
        this.height = h;
    }

    @Override
    public double getArea() {
        return (base * height)/2.0;
    }

    //possibly more Triangle methods omitted
}
```
Finally, the Refactored Class Diagram

- Final Class Diagram:
  - Normal font means the element is concrete
  - *Italicized font* means the element is *abstract*
Let's design a universal remote

- Has three buttons: up, down, reset

Can control the following devices (among other things)

- Radio (valid stations 87.5 - 107.9 MHz)
  - Increments/decrements by 0.2 MHz (and wraps around)
  - Reset goes back to 87.5

- TV volume:
  - Reset toggles mute
  - Volume has limits (let's say, 0 to 20)

- Thermostat:
  - Reset goes back to 68 degrees
public class Radio {
    public static final int MAX_FREQ = 1079;
    public static final int MIN_FREQ = 875;
    protected int station;

    public Radio() {
        reset();
    }

    public void up() {
        station += 2;
        if (station > MAX_FREQ) {
            station = MIN_FREQ;
        }
    }

    public void down() {
        station -= 2;
        if (station < MIN_FREQ) {
            station = MAX_FREQ;
        }
    }

    public void reset() {
        station = MIN_FREQ;
    }

    @Override
    public String toString() {
        return "" + (station/10.0);
    }

    //more methods omitted
public class Remote {
    // The radio that we're controlling
    protected Radio rad;

    /**
     * The constructor takes a Radio object
     */
    public Remote(Radio r) {
        rad = r;
    }

    public void pressUp() {
        rad.up();
    }

    public void pressDown() {
        rad.down();
    }

    public void pressReset() {
        rad.reset();
    }
}

More on UML:
# denotes protected
underline denotes static
Write a TV class

- **TV:**
  - 2 fields:
    - int volume (can't go past 0 and 20)
    - boolean muted
  - **up()** turns mute off and volume up by 1
  - **down()** turns mutes off and volume down by 1
  - **reset()** toggles mute (if it's on, turn mute off; if it's off, turn it on)
  - **toString()** returns a String that represents the current volume
    - "mute" if TV is muted, otherwise
    - "lllll............." (where a bar represents the current volume level)
      - (Here, volume is at 5)
public class TV {
    public static final int MAX_VOL = 20;
    protected int volume = 0;
    protected boolean muted = false;

    public void up() {
        muted = false;
        if (volume != MAX_VOL) {
            volume++;
        }
    }

    public void down() {
        muted = false;
        if (volume > 0) {
            volume--;
        }
    }

    public void reset() {
        muted = !muted;
    }

    @Override
    public String toString() {
        if (muted) {
            return "mute";
        }
        else {
            String result = "";
            for (int i=0; i<volume; i++) {
                result += "|
            }
            for (int i=0; i<MAX_VOL-volume; i++) {
                result += ".
            }
            return result;
        }
    }
}
Let's Make an Abstract Class of Devices!

- Want Remote to work with both devices without having to change code
  - **Device** is an abstract class, because `up()`, `down()`, `reset()` have the same signature, but different implementations.
    - No shared implementations means: they're all abstract methods!

<table>
<thead>
<tr>
<th>Radio</th>
<th>TV</th>
<th>Remoteable</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ final int MAX_FREQ</td>
<td>+ final int MAX_VOL</td>
<td>+ up()</td>
</tr>
<tr>
<td>+ final int MIN_FREQ</td>
<td># String name</td>
<td>+ down()</td>
</tr>
<tr>
<td># int station</td>
<td># int volume</td>
<td>+ reset()</td>
</tr>
<tr>
<td>+ up()</td>
<td># boolean muted</td>
<td>+ toString()</td>
</tr>
<tr>
<td>+ down()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ reset()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ toString()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Process Seems a Bit off This Time?

- Why does this abstract class seem a bit... off?
  - No shared fields were extracted...
  - No shared method implementations...
  - Only things that are shared are abstract methods

- The only reason we're even doing this is to have polymorphism for the Remote class. Java provides a cleaner way...
Overview

- Motivation for Even More Abstraction
- Abstract Classes
  - Interfaces
    - Abstract Classes vs. Interfaces
    - Multiple Implementations
    - Java's Comparable<T> Interface
- Conclusion
Like abstract classes, an *interface* specifies a contract between interface and an implementing class

- They are like abstract classes that only have abstract methods

Unlike abstract classes...

- The keyword *interface* is used instead of *class* in the header
- No constructors
- No fields (unless *public*, *static*, and *final* - assumed for any field)
- Abstract methods do not have the abstract keyword in signature
  - Also assumed to have *public* visibility
public interface Remoteable {
    // all fields (if any) assumed to be public static final
    // all methods assumed to be public
    void up();
    void down();
    void reset();
}

// Remoteable device
+ up()
+ down()
+ reset()

// Remote
<<interface>>
Remoteable
+ up()
+ down()
+ reset()

<<implements>>
Radio
+ final int MAX_FREQ
+ final int MIN_FREQ
# int station
+ up()
+ down()
+ reset()
+ toString()
...

<<implements>>
TV
+ final int MAX_VOL
# String name
# int volume
# boolean muted
+ up()
+ down()
+ reset()
+ toString()
...

<<implements>>
Remote
public class Radio implements Remoteable {
    public static final int MAX_FREQ = 1079;
    public static final int MIN_FREQ = 875;
    protected int station;

    public Radio() {
        reset();
    }

    @Override
    public void up() {
        station += 2;
        if (station > MAX_FREQ) {
            station = MIN_FREQ;
        }
    }

    @Override
    public void down() {
        station -= 2;
        if (station < MIN_FREQ) {
            station = MAX_FREQ;
        }
    }

    @Override
    public void reset() {
        station = MIN_FREQ;
    }

    @Override
    public String toString() {
        return "" + (station/10.0);
    }

    //more methods omitted
public class TV implements Remoteable {
    public static final int MAX_VOL = 20;
    protected int volume = 0;
    protected boolean muted = false;

    @Override
    public void up() {
        muted = false;
        if (volume != MAX_VOL) {
            volume++;
        }
    }

    @Override
    public void down() {
        muted = false;
        if (volume > 0) {
            volume--;
        }
    }

    @Override
    public void reset() {
        muted = !muted;
    }

    @Override
    public String toString() {
        if (muted) {
            return "mute";
        } else {
            String result = "";
            for(int i=0; i<volume; i++) {
                result += "|";
            }
            for(int i=0; i<MAX_VOL-volume; i++) {
                result += ".";
            }
            return result;
        }
    }
}
And the Remote?

- It's truly universal (i.e., works with polymorphic devices) now!

```java
public class Remote {
    protected Remoteable dev; // The polymorphic device that we're controlling

    public Remote(Remoteable d) {
        dev = d;
    }

    public void pressUp() {
        dev.up();
    }

    public void pressDown() {
        dev.down();
    }

    public void pressReset() {
        dev.reset();
    }
}
```
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
Abstract Classes vs. Interfaces

- **Similarities:**
  - Neither can be instantiated
    - Interfaces cannot declare constructors
    - Abstract classes can have **concrete** constructors
      - *(But what's the point if you can't instantiate abstract classes?)*
  - Both used to provide polymorphism when subclasses may have different method implementations
    - All method bodies should be inherited: Use a regular class
    - Some method bodies should be inherited: Use an abstract class
    - No method bodies should be inherited: Use an interface
Abstract Classes vs. Interfaces

- **Contrasts:**
  
  - Abstract classes may contain methods with implementations
    - So, use abstract classes if all subclasses share common behaviors
  
  - Abstract classes may contain private and protected fields
    - So, use abstract classes if all subclasses share common state
  
  - Abstract classes must be extended, interfaces must be implemented
    - Classes can only extend one subclass
    - Classes can implement multiple interfaces *(Why? Next topic)*
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
Current State

- Currently...
  - Remote works with all objects implementing Remoteable
Another Client Class

- A store that sells TVs, Radios, even universal remote controls!
  - Note: TVs and Radios still need to implement **Remoteable**
Take a Look at the Store Class

- Polymorphism: Has an ArrayList of Sellable objects
  - TVs, Radios, even Remotes are "sellable" stuff

```java
public class Store {
    protected String store_name;
    protected ArrayList<Sellable> product_line;

    public Store(String name) {
        store_name = name;
        product_line = new ArrayList<Sellable>();
    }

    public void show_product(int i) {
        if (i >= 0 && i < product_line.size()) {
            Sellable item = product_line.get(i);
            System.out.println(item.getDescription());
            System.out.println(item.getPrice());
        }
    }

    // more methods omitted...
}
```
The Sellable Interface

- Every Sellable class has to implement the following methods:
  - `getPrice()` - returns the current price of the Sellable
  - `getDescription()` - returns a String description of the Sellable

```java
public interface Sellable {
    int getPrice();
    String getDescription();
}
```
public class Remote implements Sellable {
    protected Remoteable dev; // The radio that we're controlling
    protected int price;

    /**
     * The constructor takes a Radio object
     */
    public Remote(Remoteable d) {
        dev = d;
    }

    public void up() {
        dev.up();
    }

    public void down() {
        dev.down();
    }

    public void reset() {
        dev.reset();
    }

    @Override
    public String getDescription() {
        return "blah blah";
    }

    @Override
    public int getPrice() {
        return price;
    }
}

public class Radio implements Remoteable, Sellable {
    public static final int MAX_FREQ = 1079;
    public static final int MIN_FREQ = 875;
    protected int station;
    protected boolean onsale = false;
    protected double discount = 0.15;
    protected int price;

    @Override
    public void up() {
        station += 2;
        if (station > MAX_FREQ) {
            station = MIN_FREQ;
        }
    }

    @Override
    public void down() {
        station -= 2;
        if (station < MIN_FREQ) {
            station = MAX_FREQ;
        }
    }

    @Override
    public String getDescription() {
        return "blah blah blah";
    }

    @Override
    public int getPrice() {
        if (onsale) {
            return (int) (price * (1.0 - discount));
        }
        return price;
    }
}
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
    - Java's Comparable<T> Interface
- Conclusion
Example: Comparable<T> Interface

- Interfaces allow possibly unrelated classes to implement it

- An important interface Java provides is Comparable<T>
  - Allows users to define a total ordering on objects of a certain class
    - Why? To sort your objects in some given order
  - T is the type of object that this object can be compared to.

```java
public interface Comparable<T> {

    /**
     * Compares this object with the specified object for order.
     *
     * @return a negative integer, zero, or a positive integer as this object
     * is less than, equal to, or greater than the specified object.
     */
    int compareTo(T o);
}
```
Example Comparing Shapes

- Let's say we want a way to compare shapes...
  - We'll compare shapes' areas
    - But we could very well compare by something else, like the Shape's location

Example use:

```java
Shape s1 = new Triangle(0, 0, 10, 40);  //triangle with 10 base, 40 height
Shape s2 = new Rectangle(0, 0, 10, 40);  //rectangle with 10 width, 40 height
s1.compareTo(s2)  // -200 (int)
s2.compareTo(s1)  // 200 (int)

//Two circles with radius 100
Shape c1 = new Circle(4, -10, 100);
Shape c2 = new Circle(6, 20, 100);
c1.compareTo(c2)  // 0 (int)
c2.compareTo(c1)  // 0 (int)
```
public abstract class Shape {
    protected int x;
    protected int y;

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }

    public abstract double getArea();
}
public abstract class Shape implements Comparable<Shape> {
    protected int x;
    protected int y;

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }

    @Override
    public int compareTo(Shape s) {
        return (int) (getArea() - s.getArea());
    }

    public abstract double getArea();
}
More Than Just Comparison

- It turns out that well-known Java classes exploit `compareTo()`'s ordering to sort and organize objects.

Examples:

- `Collection.sort()` -- static method to sort Collections of objects
- `Arrays.sort()` -- static method to sort arrays of objects
- Others will come later
Sorting a Collection of Objects

- What classes are Collections?
  - See [https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)

- Hey, ArrayList implements the Collection interface!

```java
import java.util.ArrayList;
import java.util.Collections;
public class Tester {
    public static void main(String[] args) {
        ArrayList<Shape> shape_list = new ArrayList<Shape>();
        shape_list.add(new Triangle(0, 0, 40, 20));
        shape_list.add(new Triangle(0, 0, 30, 20));
        shape_list.add(new Triangle(0, 0, 10, 20));
        shape_list.add(new Circle(0, 0, 50));
        System.out.println("Before sorting: "+ shape_list.toString());

        Collections.sort(shape_list);  //sort them using compareTo()
        System.out.println("After sorting: "+ shape_list.toString());
    }
}
```

- Hey, ArrayList implements the Collection interface!
Sorting a Collection of Objects (Cont.)

- Using Collections.sort() on a List.

- Output from running main() from previous slide:

Before sorting: [(0,0,40.0,20.0), (0,0,30.0,20.0), (0,0,10.0,20.0), (0,0,50.0)]
After sorting: [(0,0,10.0,20.0), (0,0,30.0,20.0), (0,0,40.0,20.0), (0,0,50.0)]
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
Conclusion

- Abstract classes and interfaces allow for even more code abstraction

- Classes can only extend one (abstract) class, but can implement multiple interfaces

- Use an **interface** when:
  
  - You realize all methods are abstract
    - That is, their exact implementation varies in subclasses
  
  - You need "multiple inheritance"
  
  - There are no instance variables (fields) to that should be inherited
Some Loose Ends: The final Keyword

- You can prevent others from the following:
  - Extending your class
  - Overriding your methods

- Need to use the final keyword:

```java
public final class Classname {
    ...
}
```

```java
public final void methodName() {
    ...
}
```
Hwk 2 posted (due next Friday 2/9)
  • Teams revealed. Let's get to know each other now.
  • 5 minutes to exchange contact info?

New code will appear on website today:
  • ShapeManager
  • ComparableShapes
Maker Space talk:
  • Johnny Devine, Head Science Teacher, SAMI
  • Tues, 2/6, 4:30-5:30
  • Library 020

Hwk 2: Get started!!!!

Lab 3 post-mortem