Recap

- Polymorphic variables (boxes) 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 "crrrrrank"

```java
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(); // crrrrrank
```
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
We want a ShapesManager class to manage a list of various shapes

- All shapes have some fields in common
  - A location (x,y coordinates), a color?, and maybe others
  - Area, but formula depends on the shape...

- Circle
  - Has a radius, r
  - Area = Math.PI * r^2

- Triangle
  - Has a base and height
  - Area = base * height / 2

- Possibly more...

- [Check out the handout for starter code]
Start with a Circle Class

```java
public class Circle {
    protected int x;
    protected int y;
    protected double radius;

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

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    public double getArea() {
        return Math.PI * Math.pow(this.radius, 2);
    }

    //possibly more methods omitted
}```
public class Triangle {
    protected int x;
    protected int y;
    protected double base;
    protected 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 this.x;
    }

    public int getY() {
        return this.y;
    }

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

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

```java
public class ShapesManager {
    protected ArrayList<Circle> shapes;

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

    public Circle getCircle(int i) {
        // is the index i valid?
        if (i >= 0 && i < this.shapes.size()) {
            return this.shapes.get(i);
        } else {
            return null;
        }
    }

    public void addCircle(Circle s) {
        this.shapes.add(s);
    }

    public double getAverageArea() {
        double area_sum = 0.0;
        for (int i = 0; i < this.shapes.size(); i++) {
            area_sum += this.shapes.get(i).getArea();
        }
        return area_sum / this.shapes.size();
    }
}
```
Practice Our New Skills! (i.e., Review)

- Current UML class diagram:

  - Class diagram after refactoring for inheritance???
    - Your turn!
    - Without writing any code
Want ShapesManager to Work with All Shapes

- Extract common fields and methods
  - Remember to intersect by functionality, not by name.
    - Thus... notice that `getArea()` wasn't retained in the superclass

<table>
<thead>
<tr>
<th></th>
<th>Circle</th>
<th>Triangle</th>
<th>Trapezoid</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields</td>
<td>int x</td>
<td>int x</td>
<td>int x</td>
<td>int x</td>
</tr>
<tr>
<td></td>
<td>int y</td>
<td>int y</td>
<td>int y</td>
<td>int y</td>
</tr>
<tr>
<td></td>
<td>double radius</td>
<td>double base</td>
<td>double base1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- double base</td>
<td>- double base1</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>+ int getX()</td>
<td>+ int get X()</td>
<td>+ int getX()</td>
<td>+ int getX()</td>
</tr>
<tr>
<td></td>
<td>+ int getY()</td>
<td>+ int get Y()</td>
<td>+ int getY()</td>
<td>+ int getY()</td>
</tr>
<tr>
<td></td>
<td>+ double getArea()</td>
<td>+ double getArea()</td>
<td>+ double getArea()</td>
<td>+ double getArea()</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

...
Yay, our **ShapesManager** accepts any shape now?

- **No! ShapesManager won't compile now! (Why?)**
- Had the same issue with `display()` in previous lecture

[Let's see the code for ShapesManager]
public class ShapesManager {
    protected ArrayList<Shape> shapes;

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

    public Shape getShape(int i) {
        // is the index i valid?
        if (i >= 0 && i < this.shapes.size()) {
            return this.shapes.get(i);
        } else {
            return null;
        }
    }

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

    public double getAverageArea() {
        double area_sum = 0.0;
        for (int i = 0; i < this.shapes.size(); i++) {
            if (i >= 0 && i < this.shapes.size()) {
                area_sum += this.shapes.get(i).getArea();
            }
        }
        return area_sum / this.shapes.size();
    }
}
Okay, we did need to define `getArea()` in superclass `Shape`

- Problem 1: How is a generic shape's `getArea()` defined?
- Problem 2: What if a subclass doesn't override `getArea()`?

---

```java
ShapesManager
- ArrayList<Shape> shapes
  + void addShape(Shape s)
  + void Shape getShape(int i)
  + double getAvgArea()
  ...

Shape
- int x
- int y
  + int getX()
  + int getY()
  + double getArea()
  ...

Circle
- double radius
  + double getArea()
  ...

Triangle
- double base
- double height
  + double getArea()
  ...

Trapezoid
- double base1
- double base2
- double height
  + double getArea()
  ...

...
...
+ double getArea()
...
```

Okay. Add `getArea()` to `Shape`. But with what implementation?
Silly to have a method that doesn't have a definition

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

    // (code omitted)

    public double getArea() {
        // Don't worry, this method shouldn't ever get called.
        // So I guess it's okay to return not-a-number (NaN)!
        return Double.NaN;
    }
}
```
Problem 2: There's no guarantee that a subclass would override it!

• (And that is a real concern. What if someone forgets to override it in a subclass?)
Another (Bad) Solution

- Leave a note and hope for the best?

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

    // (code omitted)

    public double getArea() {
        // Dear Future Coders,
        // This method just here so client classes (e.g., ShapeManager) compiles.
        // Subclasses of Shape should override this method
        // to provide specific implementations. Promise? Pretty please?? kthx!!!
        // --- With care, the Previous Coder
        return Double.NaN;
    }
}
```

This plea does not instill confidence.
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
Need a way for a superclass to:

- *Convince* Java to compile without some methods written in the superclass, *BUT*....
- *Demand* that those methods be defined in the subclass(es).

How? Hot new Java keyword-of-the-day: `abstract`

- Only used in two places:
  - In an *abstract class* declaration
    - Usually named "AbstractClassname" (not required, just convention)
    - Like AbstractShape, AbstractList, *etc.*
  - In an *abstract method* declaration
Abstract Classes

```java
public abstract class AbstractClassName {
    // instance variables declared here just like before
    // ...

    // constructors declared here just like before
    // ...

    // methods declared below just like before
    // ...
}
```
Abstract Classes

public abstract class AbstractClassName {
    // instance variables declared here just like before
    // ...

    // constructors declared here just like before
    // ...

    // methods declared below just like before
    // ...

    // abstract methods may be declared below... (no implementations)
    public abstract void y();
    public abstract int z(String s);
}

Specifying a Contract (Cont.)
Abstract Class Properties (Know These)

- Characteristics of Abstract Classes
  - How they're like regular ol' classes:
    - Can have public/protected/private fields
    - Can have methods
    - Can have constructors
  - How they're different from classes:
    - Objects of abstract classes \textit{cannot} be instantiated!
      - (What's a point of having constructors then?)
    - Can contain methods without bodies:
      - These \textit{abstract methods} \textit{must be overridden} by subclasses
        » This is the "contract" between an abstract class and its subclass(es)
```java
public abstract class AbstractShape {
    protected int x;
    protected int y;

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

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    // This is just here so client classes that use AbstractShape can compile!
    // Contract: subclasses must override and provide their own implementation
    public abstract double getArea();
}
```
Shape as an Abstract Class

```java
public abstract class AbstractShape {
    protected int x;
    protected int y;

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

    // This is just here so client classes that use AbstractShape can compile!
    // Contract: subclasses must override and provide their own implementation
    public abstract double getArea();
}
```

- The **only** purpose for abstract classes is for subclasses to extend it.

```java
AbstractShape s = new AbstractShape(0,0); //compiler error
```
A Concrete Triangle Class

```java
public class Triangle extends AbstractShape {
    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 (this.base * this.height)/2.0;
    }

    // (possibly more Triangle methods omitted)
}
```
public class Circle extends AbstractShape {
    protected double radius;

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

    @Override
    public double getArea() {
        return Math.PI * Math.pow(this.radius, 2);
    }

    // (possibly more Circle methods omitted)
}
What if we \textit{didn't} implement an abstract method?

\begin{verbatim}
public class Circle extends AbstractShape {
    protected double radius;

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

Contract breached! Gives the following,
Compiler error: The type Circle must implement the inherited abstract method: getArea()
Finally, the Refactored Class Diagram

- **Final Class Diagram:**
  - Normal font means the element is concrete
  - *Italicized font* means the element is *abstract*
public class ShapesManager {
    protected ArrayList<AbstractShape> shapes;

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

    public AbstractShape getShape(int i) {
        // is the index i valid?
        if (i >= 0 && i < this.shapes.size()) {
            return this.shapes.get(i);
        }
        else {
            return null;
        }
    }

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

    public double getAverageArea() {
        double area_sum = 0.0;
        for (int i = 0; i < this.shapes.size(); i++) {
            area_sum += this.shapes.get(i).getArea();
        }
        return area_sum / this.shapes.size();
    }
}