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**
  - Both are used to provide polymorphic variable types
    - So *client classes* can work with "many shapes"
    - *e.g.*, Farm, Remote, Library (lab 3)
  - Neither can be instantiated
  - Both are used when subclasses have different method implementations
Abstract Classes vs. Interfaces

- Differences

  - Interfaces **can't** inherit/extend from a super-interface
    - Abstract classes **can** (the Abstract LibraryBook extends from Book in lab 3)
  
  - Interfaces **don't** have private and protected fields
  
  - Interfaces **don't** have constructors
  
  - Interfaces **don't** contain methods with implementations
  
  - Another huge difference
    - Subclasses can only extend **one** superclass
    - Subclasses can implement **multiple** interfaces
      
      - *(Why would you want to? Next topic)*
When to Use ... ?

- As a software designer you need to make important design decisions

- When you need polymorphic types for a client class,
  - **Provide a run-of-the-mill super class**... *(e.g., Animal, Mammal)*
    - When fields and methods should be inherited
  - **Provide an abstract super class**... *(e.g., LibraryBook, AbstractShape)*
    - When fields and *some* methods should be inherited
    - When the implementation of some methods aren't clear, but the method signature must exist for compiling client classes
      - What's the *getArea* of just Shape? What's the *circulationStatus* of a lib book?
  - **Provide an interface**... *(e.g., Controllable, Sellable)*
    - When no fields or methods should be inherited (every method is abstract)
Overview

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

- Remote works with all objects implementing Controllable
Let's Add Another Client Class: Store

- What if we wanted a **Store** that sells these items?
  - Sells TVs, Radios, even universal remote controls, and more!
  - Store needs a polymorphic list
Let's Add Another Client Class (Cont.)

```
show_product(int i)
...
```

<table>
<thead>
<tr>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td># String store_name</td>
</tr>
<tr>
<td># ArrayList&lt;Sellable&gt; product_line</td>
</tr>
</tbody>
</table>

```
<<interface>>
Sellable
+
getDescription()
+ getPrice()
...```

```
<<interface>>
Controllable
+ up()
+ down()
+ reset()
...```

```
Radio
+ final int MAX_FREQ
+ final int MIN_FREQ
# int station
+
up()
+ down()
+ reset()
+ getDescription()
+ getPrice()
+ toString()
...```

```
TV
+ final int MAX_VOL
# int volume
# boolean muted
+
up()
+ down()
+ reset()
+ getDescription()
+ getPrice()
+ toString()
...```

```
Remote
# Controllable device
+ pressUp()
+ pressDown()
+ pressReset()
+ toString()
...```

```
<<interface>>
Sellable
+
getDescription()
+ getPrice()
...```

```
<<interface>>
Controllable
+ up()
+ down()
+ reset()
...```

```
Remote
# Controllable device
+ pressUp()
+ pressDown()
+ pressReset()
+ toString()
...```

```
<<interface>>
Sellable
+
getDescription()
+ getPrice()
...```

```
<<interface>>
Controllable
+ up()
+ down()
+ reset()
...```

```
Radio
+ final int MAX_FREQ
+ final int MIN_FREQ
# int station
+
up()
+ down()
+ reset()
+ getDescription()
+ getPrice()
+ toString()
...```

```
TV
+ final int MAX_VOL
# int volume
# boolean muted
+
up()
+ down()
+ reset()
+ getDescription()
+ getPrice()
+ toString()
...```

```
Remote
# Controllable device
+ pressUp()
+ pressDown()
+ pressReset()
+ toString()
...```

```
<<interface>>
Sellable
+
getDescription()
+ getPrice()
...```

```
<<interface>>
Controllable
+ up()
+ down()
+ reset()
...```

```
Radio
+ final int MAX_FREQ
+ final int MIN_FREQ
# int station
+
up()
+ down()
+ reset()
+ getDescription()
+ getPrice()
+ toString()
...```

```
TV
+ final int MAX_VOL
# int volume
# boolean muted
+
up()
+ down()
+ reset()
+ getDescription()
+ getPrice()
+ toString()
...```
Every Sellable class has to implement the following methods:

- `getPrice()` - returns the current price of the Sellable item
- `getDescription()` - returns a String description of the Sellable item

```java
/**
 * Classes implementing this interface can be sold
 * and must provide getPrice and getDescription methods.
 * @author David
 */
public interface Sellable {
    /**
     * @return the current price of the item
     */
    int getPrice();

    /**
     * @return a description of the item
     */
    String getDescription();
}
```
public class Remote implements Sellable {
    protected Controllable dev;       // The radio that we're controlling
    protected int price;

    /**
     * The constructor takes any "Controllable" object
     */
    public Remote(Controllable d) {
        this.dev = d;
    }

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

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

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

    @Override
    public String getDescription() {
        return "The best universal remote";
    }

    @Override
    public int getPrice() {
        return this.price;
    }
}
public class Radio implements Controllable, 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;

    public Radio() {
        this.reset();
    }

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

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

    @Override
    public String getDescription() {
        return "This is a great radio.";
    }

    @Override
    public int getPrice() {
        if (this.onsale) {
            return (int) (this.price * (1.0-this.discount));
        }
        return this.price;
    }
}
Polymorphism: A Store has an ArrayList of "Sellable" items

- TVs, Radios, Remotes are all "Sellable"

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

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

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

    // more methods omitted...
}
```
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
AbstractShape s1 = new Triangle(0, 0, 10, 40);  //triangle with 10 base, 40 height
AbstractShape s2 = new Rectangle(0, 0, 10, 40);  //rectangle with 10 width, 40 height
s1.compareTo(s2) > -1  //int
s2.compareTo(s1) > 1   //int

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

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

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

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

    @Override
    public int compareTo(AbstractShape s) {
        if (this.getArea() == s.getArea()) {
            return 0;
        }
        else if (this.getArea() < s.getArea()) {
            return -1;
        }
        return 1;
    }

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

- It turns out that well-known Java classes exploit `compareTo()`'s ordering to sort objects by their "natural ordering."

- How to sort Objects? Here are some ways:
  - `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

- 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<AbstractShape> shape_list = new ArrayList<>();
        shape_list.add(new Triangle(0,0,40,20));
        shape_list.add(new Circle(0,0,50));
        shape_list.add(new Triangle(0,0,30,20));
        shape_list.add(new Triangle(0,0,10,20));

        System.out.println("Before sorting: \n" + shape_list.toString());
        Collections.sort(shape_list); //sort the shapes using compareTo()
        System.out.println("After sorting: \n" + shape_list.toString());
    }
}
```
Sorting a Collection of Objects (Cont.)

- Using Collections.sort() on a List.
- Output from running main() from previous slide:

Before sorting: 

\[(0,0,40.0,20.0), (0,0,30.0,20.0), (0,0,10.0,20.0), (0,0,50.0)\]
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), (0,0,50.0), (0,0,30.0,20.0), (0,0,10.0,20.0)]
After sorting: [(0,0,10.0,20.0), (0,0,30.0,20.0), (0,0,40.0,20.0), (0,0,50.0)]
Classtime Activity!

- Our Library from lab (**handouts** to come)
  - But we weren't never sorted the books in the library by call number!

- Your task:
  - Add method `organize()` to Library
    - It sorts the books by their call number
  - Necessitates `LibraryBook` to implement `Comparable`
Overview

- Motivation for Even More Abstraction
- Abstract Classes
- Interfaces
  - Abstract Classes vs. Interfaces
  - Multiple Implementations
  - Java's Comparable<T> Interface
- Conclusion
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() {
    ...
}
```
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
Hwk 2 posted (due next Wed 9/25)

- Teams have been set
- Let's get to know each other now.
  - 2 minutes to exchange contact info.
- Opportunity for bonus points

Today

- The need for more abstraction
- Superclass and Subclass Contracts
- The `abstract` keyword
Grad school panel (for Math, Physics, CS)
  - Monday, 9/23
  - 4pm in TH 395

Hwk 2 due next Wed 9/25

Lab 3 post-mortem
  - Abstract classes can extend other classes too!
  - Working with abstract classes, method overriding
  - Finish it on your own for practice
Administrivia 9/20

- Grad school panel (for Math, Physics, CS)
  - Monday, 9/23
  - 4pm in TH 395

- Hwk 2 due next Wed 9/25

- Last time...
  - Ternary operator
  - Unpacked Lab 3
  - When do we use an Interface?
Hwk 1 graded
  • Median: 75
  • Common issues:
    - null input
    - Extraneous code
    - Lack of comments
  • Optional regrade
    - Earn back half of points missed if resubmitted by end of the week
Hwk 2 due Wednesday
Lab 3 tomorrow on Interfaces