Overview

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

- Can you succinctly articulate the following?
  - When do you use abstract classes over regular classes?
  - Why do you *need* abstract methods?
  - What's the point of having constructors in abstract classes?
  - When should abstract classes extend a superclass?
    - (Like the abstract LibraryBook class from Lab)
Let's design a really simple *universal remote*

- Only has 3 buttons to press: 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
Let's Write a TV class

- A TV class has
  - int volume (can't go past 0 and 20)
  - boolean muted
  - `up()` turns mute off, and volume++
  - `down()` turns mutes off, and volume--
  - `reset()` toggles mute (if it's on turn it off; if it's off turn it on)
  - `toString()` returns a String that represents the TV volume
    - "mute" if TV is muted
    - Otherwise, "llllllllllllllllllll" (bars = current volume)
      - (Here, volume is at 5)
Now Introduce TVs

```java
public class TV {
    public static final int MAX_VOL = 20;
    protected int volume;
    protected boolean muted;

    public TV() {
        this.volume = 0;
        this.muted = false;
    }

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

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

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

@Override
public String toString() {
    if (this.muted) {
        return "mute";
    }
    //construct volume bar
    String result = "";
    for (int i = 0; i < MAX_VOL; i++) {
        if (i < this.volume) {
            result += "|";
        } else {
            result += ".";
        }
    }
    return result;
}
```
Aside: Ternary Operator

- A convenient short hand
  - Used when assigning a variable value that depends on the result of a condition
  - Syntax: \( \text{var} = (\text{cond}) ? \text{value1} : \text{value2}; \)

- The assignment operator = can also be: += or -=

- Syntactically equivalent to:

```java
if (cond) {
    var = value1;
}
else {
    var = value2;
}
```
Aside: Ternary Operator

- A convenient short hand
  - Used when assigning a variable value that depends on the result of a condition
  - Syntax: \( \text{var} = (\text{cond}) \ ? \ \text{value1} : \ \text{value2}; \)
    - The assignment operator = can also be: += or -=

Examples:

```java
Random rng = new Random();
String coin = (rng.nextDouble() < 0.5) ? "head" : "tail";

int x = getNumFromUser();
String str = "Your number is ";
str += (x % 2 == 0) ? "even" : "odd";
System.out.println(str);
```
Now Introduce TVs (with Ternary Op)

```java
public class TV {
  public static final int MAX_VOL = 20;
  protected int volume;
  protected boolean muted;

  public TV() {
    this.volume = 0;
    this.muted = false;
  }

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

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

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

  @Override
  public String toString() {
    if (this.muted) {
      return "mute";
    }
    //construct volume bar
    String result = "";
    for (int i = 0; i < MAX_VOL; i++) {
      result += (i < this.volume) ? "|" : ".";
    }
    return result;
  }
}
```
Let's Checkout Radios First

```java
public class Radio {
    public static final int MAX_FREQ = 1079;
    public static final int MIN_FREQ = 875;
    protected int station;

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

    public void up() {
        this.station += 2;
        // wrap around to beginning?
        if (this.station > MAX_FREQ) {
            this.station = MIN_FREQ;
        }
    }

    public void down() {
        this.station -= 2;
        // wrap around to end?
        if (this.station < MIN_FREQ) {
            this.station = MAX_FREQ;
        }
    }

    public void reset() {
        this.station = MIN_FREQ;
    }

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

//more methods omitted
Our Client Class "Remote" Needs Refactored

- Currently, our Remote only works with Radios

```java
public class Remote {
    // The radio that we're controlling
    protected Radio rad;

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

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

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

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

More on UML:
- **underline** denotes static
Want **Remote** to work with both devices without having to change code

- **Controllable** is an abstract class, because `up()`, `down()`, `reset()` have the same signature, but different implementations.
  - No shared implementations means: they're all **abstract**!

<table>
<thead>
<tr>
<th>Radio</th>
<th>TV</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ final int MAX_FREQ</td>
<td>+ final int MAX_VOL</td>
</tr>
<tr>
<td>+ final int MIN_FREQ   # int station</td>
<td># String name</td>
</tr>
<tr>
<td></td>
<td># int volume        # boolean muted</td>
</tr>
<tr>
<td>+ up()</td>
<td>+ up()</td>
</tr>
<tr>
<td>+ down()</td>
<td>+ down()</td>
</tr>
<tr>
<td>+ reset()</td>
<td>+ reset()</td>
</tr>
<tr>
<td>+ toString()</td>
<td>+ toString()</td>
</tr>
</tbody>
</table>

\[ \text{Radio} \cap \text{TV} = \text{Controllable} \]

\[
= \begin{array}{c}
\text{Radio} \\
\text{TV} \\
\text{Controllable}
\end{array}
\]

\[
\begin{align*}
\text{Radio} &= \begin{array}{l}
\text{Radio} \\
\text{TV}
\end{array} \\
\text{Controllable} &= \begin{array}{l}
\text{Radio} \\
\text{TV}
\end{array}
\end{align*}
\]
This Abstract Class Doesn't Seem Satisfying

Why does this abstract class seem off?

- No shared instance variables...
- No shared methods...
- So, what are subclasses even inheriting?

The only reason we're even doing this is to have a polymorphic type in the universal Remote client class.
Overview

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

- Interfaces are like abstract classes that only have abstract methods

**Unlike** abstract classes...

- No constructors allowed
- Fields automatically assumed *public*, *static*, *and* *final*!
- Abstract methods assumed to have *public* visibility
  - (So *public*, *abstract* keywords are superfluous)
Specifying an Interface

- Interface syntax

```java
public interface InterfaceName {
    // All fields assumed to be public static final
    // ... list them here (if any)
    double PI = 3.14159;

    // Abstract methods declared below
    // (public visibility is assumed)
    void y();
    int z(String s);
    double w(double x);
}
```
The following isn't *wrong*, just superfluous due to assumptions

- This will compile, but it would be bizarre to see in real world

```java
public interface InterfaceName {
    // All fields assumed to be public static final
    // ... list them here (if any)
    public static final double PI = 3.14159;

    // Abstract methods declared below
    // (public visibility is assumed)
    public abstract void y();
    public abstract int z(String s);
    public abstract double w(double x);
}
```
Since interfaces are not by definition classes, new classes do not extend Interfaces!

- They implement interfaces. **Syntax:**

```java
public class ClassName implements InterfaceName {
  // fields
  // constructors
  // methods

  // must override abstract methods
  @Override
  public void y() {
    // (code omitted)
  }

  @Override
  public int z(String s) {
    // (code omitted)
  }

  @Override
  double w(double x) {
    // (code omitted)
  }
}
```
public interface Controllable {
/**
 * Defines the "up" function on the device
 */
 void up();

/**
 * Defines the "down" function on the device
 */
 void down();

/**
 * Defines the "reset" function on the device
 */
 void reset();
}
Radio Implementation (Sorry no Javadocs)

```java
public class Radio implements Controllable {
    public static final int MAX_FREQ = 1079;
    public static final int MIN_FREQ = 875;
    protected int station;

    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 void reset() {
        this.station = MIN_FREQ;
    }

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

    //more methods omitted
}
```
public class TV implements Controllable {
    public static final int MAX_VOL = 20;
    protected int volume;
    protected boolean muted;

    public TV() {
        this.volume = 0;
        this.muted = false;
    }

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

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

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

    @Override
    public String toString() {
        if (this.muted) {
            return "mute";
        }
        //construct volume bar
        String result = "";
        for (int i = 0; i < MAX_VOL; i++) {
            result += (i < this.volume) ? "|" : ".";
        }
        return result;
    }
}
And the Remote (Client Class)?

- It's truly universal now \textit{(i.e., works with polymorphic devices)!}
  - \textit{Any} device that implements Controllable can be input to constructor!

```java
public class Remote {
    protected Controllable device;  // A polymorphic device we're currently controlling

    public Remote(Controllable d) {
        this.device = d;
    }

    public void pressUp() {
        this.device.up();
    }

    public void pressDown() {
        this.device.down();
    }

    public void pressReset() {
        this.device.reset();
    }
}
```