Panel 1

Last time:

**Classes**: blueprint to define new types
- fields (store data) + methods (to do stuff)
- constructor (class name, no return type)

**Objects**: instantiated from classes via "new"
- only exist in Memory for read
- calls constructor

Panel 2

Data Encapsulation: protect integrity of a class
- private: only objects of same class can access
- public: everyone can access
- protected: everyone in same package or same (hidden) directory has access

Only applies to fields/methods, not local vars

Rule: fields should be private
- methods could be public
- be as restrictive as possible
Panel 3

<table>
<thead>
<tr>
<th>Clear Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>(built-in) has: value and unit ( \frac{100 ^\circ C}{100 K} )</td>
</tr>
<tr>
<td>do: convert to ( ^\circ C, \text{K}, ^\circ F )</td>
</tr>
<tr>
<td>add / subtract / display</td>
</tr>
<tr>
<td>ensure integrity of Temp.</td>
</tr>
</tbody>
</table>

```java
class Temperature {
    double value;
    char unit;

    Temperature(double value, char unit) {
        TemperatureT();
    }

    add();
}
```

Panel 4

```java
public class Temperature {
    private double value = 0.0;
    private double unit = 'C';

    public Temperature(double value, double unit) {
        public TemperatureT();
    }

    public TemperatureT( ) (etc.)

    public Temperature add (Temperature +) (etc.)
}
```
Panel 5

Constructor

    public Temperature (double value, char unit)
    {
        this.value = value;
        if ((unit == 'F') || (unit == 'f'))
            this.unit = 'F';
        else if ((unit == 'K') || (unit == 'k'))
            this.unit = 'K';
        else if ((unit == 'C') || (unit == 'c'))
            this.unit = 'C';
        else
            this.unit = 'I';
    }

Panel 6

public void display()
{
    if (unit != 'I')
        System.out.println(value + "" + unit);
    else
        System.out.println("invalid temp");
}

Poorly designed method because it assumes text-based output, instead

public String display() 
{ 
    return "b String";
} 

return 

Panel 7

1. Write program to convert
   100°C to °F
   \[ 100 \text{°C} \rightarrow 212\text{°F} \]

2. Add "add" and "subtract" methods to home!
   Say I decide to: add always in the `°C`
   \[
   +\text{add}(+2), \text{to Fahrenheit}()
   \]

3. Really want to show output in
   text fields

Panel 8

The "static" modifier

fields (and methods) can be static or not.
If non-static, every object gets its own copy of the field
If static, every object shares the same field

Static methods can only refer to static data

Rule of thumb: Static method gets all input from input variables or local variables.
Panel 9

```java
public class Temperature {
    double value;

    public Temperature(double value) {
        this.value = value;
    }

    public static double toFahrenheit(double celsius) {
        return (celsius * 9/5) + 32;
    }
}
```

Panel 10

Static methods are often "computational". They can be used without "new" but by giving class Name.

```java
class
    double x = Math.sin(Math.PI);
```

Commonly used static methods:

```java
Math.sin(0), Math.cos(0), ...
OptionPane.showMessageDialog(null, "...");
```
import javax.swing.JOptionPane;

public class TempTest {
    public static void main(String args[]) {
        JOptionPane.showMessageDialog(null, "Enter temperature");
        Temperature t = new Temperature(100.0, 'C');

        String s = t.toString() + "n";
        s += " = " + t.toFahrenheit().toString() + "F";
        s += " = " + t.toKelvin().toString() + "K";
        s += " = " + t.toCelsius().toString() + "C";
        JOptionPane.showMessageDialog(null, s);
    }
}